Automating UML Models Refactoring using Search-Based Algorithms

BY

Abdulrahman Ahmed Bobakr Baqais

A Dissertation Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

May 2016

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by Abdulrahman Ahmed Bobakr Baqais under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.**

Dr. Mohammad Alshayeb
(Advisor)

Dr. Adel Fadhl
CCSE College Dean

Dr. Shorki Selim
(Member)

Dr. Salam A. Zummo
Dean of Graduate Studies

Dr Farag Azzedin
(Member)

14/11/16
Date

Dr. Mahmoud Niazi
(Member)

Dr. Muhammad Elrabaa
(Member)

To my father, mother, brothers, Maymoonah and Razan

ACKNOWLEDGMENTS

I would like to thank first and foremost the almighty Allah, who gives me the brain to think, the hands to write, the faith to cope up with all obstacles. I declare that any and all achievements I accomplished in my life are due to his almighty support.

I would like to thank my father and mother who regardless of their financial situation, their age, and their concern of other life matters never ever stopped supporting me to become the man I am now. No word, money, or award can reimburse a period spans more than a decade of support. I ask Allah to award my passing away father in the hereafter and award my mother in this life and in the hereafter.

In line with that, I would like to thank my elder brother who spent all of his money on my education without remorse, regret or asking for return.

I would like to thank my wife who shares with me the bitterness of PhD journey. Awake through nights, staying with me in the weekend, traveling from one place to another and allowing me to work on my thesis hours and hours alone. This piece of work might be done by my hand, but surely, it is done by her sacrificing.

I would like to thank Dr. Salam Zummo the Dean of the graduate studies who listened to me carefully and gave me another chance to start another dissertation with a new supervisor. His trust in me is invaluable.

I would like to thank my supervisor Dr. Alshayeb. He is such a good advisor. He is so good and he helps a lot. He always pushes me to do better and better. No one can have such an advisor. He talks to me about other things rather than dissertation sometimes. He

encouraged me to work towards publications. He is really a good advisor. He meets me every week once or more just to ensure the work is done. He is working so hard to supervise me and monitor my work. He likes to share his experience in the US and how he struggled and suffered there. Thanks to my committee members: Prof Dr. Shokri Selim, Dr. Niazi, Dr. Mohammaed Elrabaa and Dr. Farag for their time and support too.

A special thanks to the chairman Dr. Abdulaziz Alkhoraidly, who is genuinely born as a leader and his words inspires and flourishes in front of me whenever I feel the sorrow of PhD ordeal.

Thanks to my undergraduate students who always like to listen to my PhD stories.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	:	Artificial Intelligence
BBN	:	Bayesian Belief Network
BPN	:	Back propagate on Neural Network
DT	:	Decision Tree
GEP	:	Genetic Expression Programming
HC	:	Hill Climbing
KNN	:	K Nearest Neighbour
LACE	:	Late Acceptance Hill Climbing
LR	:	Logistic Regression
MLP	:	Multi-layer Perceptron
NB	:	Naïve Bayes
NN	:	Neural Network
OO	:	Object Oriented
PNN	:	Probabilistic Neural Network
RBF	:	Radial Basis Function

SA	:	Simulated Annealing
SVM	:	Support Vector Machine
UML	:	Unified Modeling Language
XML	:	eXtensible Modeling Language

ABSTRACT

Full Name : Abdulrahman Ahmed Bobakr Baqais

Thesis Title : Automating UML Models Refactoring using Search-Based Algorithms

Major Field : Computer Science and Engineering

Date of Degree : May 2016

Refactoring is intended to improve the software quality by detecting defects in the software and correcting it. This process reduces the effort and cost of maintenance which is reported to be the most expenditure activity in the software development process. To refactor, there are different approaches: manual, semi-automated and automated.

Automatic refactoring has been approached at the code level of the software. However, there is a scarcity of research of applying automatic refactoring at UML models. Design usually precedes coding activity and as such correcting any defects early will save the time, cost and effort of testing and maintaining the software.

The objective of this dissertation is to use different types of algorithms such as: search based and machine learning to compare the refactoring process on different UML diagrams such as: use case diagram, sequence diagram and class diagram. In addition, we implemented search based algorithms on an integrated multiple-view model that composes the three aforementioned diagrams. To validate that our automatic refactoring is meaningful and beneficial, a set of quality metrics must be improved. An empirical validation of these approaches on different case studies reveal many interesting results, issues, challenges of applying AI for UML diagram refactoring. Simulated Annealing (SA) performed the best comparing to Hill Climbing (HC) and Late Acceptance Hill

Climbing (LAHC) in refactoring use case diagram. A hybridization of K-means and Simulated Annealing (KSA) was able to find all the refactoring opportunities in sequence diagram. Support Vector Machine (SVM) was able to perform equally competitive in training and validation phase in finding class diagram smells. Again, SA performed very well comparing to HC and LAHC in detecting and refactoring a multiple-view model composing of the three selected models. The algorithms were able to achieve high accuracy and recall of detecting anti-patterns or defective components in different UML diagrams.

ملخص الرسالة

الاسم الكامل: عبدالرحمن احمد بوبكر باقيس

عنوان الرسالة: أتمتة اعادة هيكلة نماذج لغة النمذجة الموحدة باستخدام الخوارزميات المعتمدة على البحث

التخصص: علوم وهندسة الحاسب الالي

تاريخ الدرجة العلمية: مايو 2016

الغرض من اعادة هيكلة البرمجيات هو تطوير جودة البرنامج عن طريق ايجاد الاخطاء الموجودة فيه وتصحيحها. تقلل هذه العملية من تكلفة اصلاح البرنامج والجهد المبذول والذي تقول الاحصائيات انه اكثر فترة مكلفة في مراحل تطوير البرنامج. هناك العديد من الطرق لا عادة هيكلة البرمجيات مثل: الطرق اليدوية، نصف الاتوماتيكية والطرق الاتوماتيكية.

تم تنفيذ الكثير من اعادة الهيكلة الاتوماتيكية على الأكواد البرمجية فقط. ولكن هناك نقص واضح في الابحاث المتعلقة بتطبيق اعادة الهيكلة الاتوماتيكية على نماذج لغة النمذجة الموحدة. التصميم عادة يسبق كتابة الأكواد ولذلك فإن تصحيح الاخطاء مبكرا سيوفر الكثير من المال والجهد والوقت في فترة اختبار او صيانة البرنامج.

الهدف من هذه الرسالة هو استخدام عدة خوارزميات من خوارزميات الذكاء الاصطناعي لإعادة هيكلة عدة نماذج من لغة النمذجة الموحدة. هذه النماذج تشمل : رسم بياني لحالة الاستخدام ، رسم بياني للتسلسل ، رسم بياني للأصناف. بالإضافة الى ذلك نحن طبقنا خوارزميات الذكاء الاصطناعي على نموذج متعدد الاوجه ومتكامل يشمل جميع الرسوم البيانية السابقة. لضمان ان اعادة الهيكلة ذات قيمة ومعنى فان اعادة الهيكلة لابد ان تحسن من بعض معايير الجودة المتعلقة بالرسم البياني.

أظهرت المصادقة العملية ا على اكثر من حالة دراسية الكثير من النتائج المثيرة بالإضافة الى بعض الاشكاليات والتحديات من تطبيق الخوارزميات المعتمدة على البحث على نماذج لغة النمذجة الموحدة. استطاعت خوارزمية التخمير المحاكي من الحصول على أفضل أداء مقارنة بخوارزمية تسلق التل و خوارزمية القبول المتأخر لتسلق التل

لإعادة هيكلة نموذج حالة الاستخدام. استطاعت الخوارزمية المدمجة من خوارزمية التخمير المحاكي والخوارزمية التصنيفية ايجاد جميع فرص اعادة الهيكلة في النموذج التسلسلي. استطاعت خوارزمية شعاع الدعم الآلي من تحقيق كفاءة متساوية متنافسة في مرحلتي التدريب والاختبار في ايجاد الروائح الكريهة في نموذج الأصناف. استطاعت خوارزمية التخمير المحاكي مرة أخرى من الحصول على أفضل أداء مقارنة بخوارزمية تسلق التل و خوارزمية القبول المتأخر لتسلق التل. استطاعت هذه الخوارزميات من الحصول على نتائج جيدة من ناحية الدقة والاسترجاع لكثير من الاخطاء المتعلقة في التصميم في مختلف الرسوم البيانية من لغة النمذجة الموحدة.

CHAPTER 1

INTRODUCTION

Refactoring tends to improve the internal structure of the software while preserving its behaviour [3]. This process attempts to reduce the complexity of the software and cut its maintaince cost [4] promoting its quality status [5]. Refactoring tends to improve readability, understandability, maintainability, portability (reusability) and others.

Software quality implies different attributes and characteristics to different stakeholders. In large systems, it becomes very hard to determine the necessary quality factors. Clients opt for running software with higher performance. Testers opt for testability feature of the code, programmers prefer reusability and understandability, project managers focus on the cost..etc. Tradeoff is a resolution that all are agreeing upon, but the question is what to sacrifice and what to optimize and how to ensure its impact on the quality. Thus, refactoring takes several forms and initiates different actions to serve the above purposes. The surge of literature addressing refactoring issues can be observed in the last few years and though many are dedicated to code refactoring, this proposal is addressing design refactoring

1.1 Problem Statement

The majority of articles that discuss software refactoring are focusing on software code [3, 6, 7]. Recently, a slight increase in the interest of refactoring at the design level emerged [8]. Different methods have been applied to refactor UML diagrams namely: pattern-based [9], formal rules [10] and graph transformation [11].

Refactoring UML diagrams is favorable since designing activity precedes coding and as such abnormalities, ill-structure or potential bugs can be detected and corrected early [12]. Each UML diagram has different design smells and requires different refactoring operations.

From the various approaches to perform UML refactoring, very few targeted the advantatges, transperancy and performance that Artificial Intelligence(AI) techniques might import to the refactoring proces [13]. Some of the issues when applying AI techinques for model refactoring (design-level refactoring) include: encoding method. Other issues include: adjusting the algorithm to the structure of the design and adjusting the algorithm to the objective functions (quality metrics). These usually are the major issues that face any developer in developing an AI algorithm to a UML diagram.

In [13], they used interactive genetic algorithm for a class diagram refactoring. Our approach differs in different ways. We used different algorithms, tested on different diagrams representing different views including a multiple-view diagram. In addition, they used diagrams reversed from code while we worked on real diagram models. Our

algorithm is totally automated requires no intervention from the users, while their algorithm depends on an interactive response from the user.

Refactoring UML design manually exhibits some drawbacks such as: it's costly in terms of cost and time and it requires domain experts. Automating the refactoring process surely will save time and cost and will help software practioners to improve their designs.

Most of the other refactoring approaches are carried out on single instances of UML diagrams [14, 15]. In this research, we are extending the field by applying AI refactoring on a multi-view UML model and comparing the results with individual UML diagrams. This will show the advantages of adopting multi-view refactoring since a multiple-view model usually gives a more information about a refactoring opportunity.

The overall aim of this research is to "*refactor UML models including a multiple-view model by providing the user with a set of AI techniques, that utalize software metrics and refactoring operations, to produce refactoring sequences that improve quality*".

1.2 Motivation

This research started with a background survey on the different smells of UML models , the required operations to eliminate these smells and the right metrics that measure various dimensions of model quality and followed by a brief desctpion of various AI techniques that were applied in refactoring at source code and design level. However, most of the recent articles in the literature focus on source-code level refactoring, this research seeks the developemnt of applying AI refactoring at the design level to various UML diagrams. It sets itself apart from other works by including a multi-view UML

model (A novel model proposed by a graduated PhD student of the department [16]) and refactor it along with a detailed comparison between UML diagrams refactoring and multi-view UML refactoring utilizing different AI algorithms which were not applied for refactoring before.

Overall, the contributions of this dissertation can be observed from these angles: 1) Automatic detection and refactoring operation of various UML diagrams including a multi-view model. 2) Employing and analyzing AI refactoring at a single and multi-view UML model and 3) Realizing model metrics that impacts positively on the whole quality of the model.

A recent thoroughly systematic review published in Empirical Software Engineering Journal emphasizes the need to pursue further research on UML Model Refactoring: "*The results of this review indicated that UML model refactoring is a highly active area of research. Quite a few quality techniques and approaches have been proposed in this area, but it still has some important issues and limitations to be addressed in future work"* [8].

In summary, our proposed approach of refactoring is in an alignment with the following quotes by O'Keeffe and Ó. Cinnéide: "Recently, Artificial Intelligence approaches to automating the task of software refactoring, based on the concept of treating object-oriented design as a combinatorial optimization problem, have been proposed. … and are inspired by the successful application of Artificial Intelligence approaches in other areas of software engineering…" [6].

1.3 Organization

The structure of this dissertation is organized as follows: Chapter 2 presents a background on UML and its different views, refactoring in general, metrics, transformation and how refactoring can be seen as an optimization problem. Chapter 3 provides an extensive literature review about the two components of our research: refactoring whether at code level or at model level, techniques and related works where these techniques have been applied for refactoring problems. Chapter 4 is dedicated to the research questions, objectives and research methodology. The chapter also provides a sketch diagrams on the processes involved in accomplishing our objectives. At the end of this chapter, we describe the solution flow with a brief explanation on each step. Chapter 5 shows the methodology of our work on use case diagram. Chapter 6 is for sequence diagram. Chapter 7 is dedicated to the class diagram and chapter 8 shows the applicability of our approach to a multiple-view diagram. Finally, chapter 9 concludes the dissertation with an emphasis on the contributions and limitations of our research. A few appendices are compiled at the end serving as a reference for this work.

CHAPTER 2

BACKGROUND

This background is composed of six components that represent different domain knowledge on which the research idea is built upon: UML models, Refactoring, Transformation, Metrics, Techniques (AI specifically) and optimization.

2.1 Unified Modeling Language (UML)

UML is a modeling language targeting Object-Oriented (OO) Paradigm. It was developed by Grady Booch, Ivar Jacobson and James Rumbaugh in 1996 and adopted later by the Object Management Group (OMG). It has two versions UML 1.0 and UML2.0 with several minor releases in between. Mainly, this modeling language is composed of three views namely: Structural, Behavioral and Interaction [1].

Structural View is concerned with the overall components of the system and how they are stacked, aligned or grouped. It is composed of the following Diagrams: "*Profile Diagram*", "*Class Diagram*", "*Composite Structure Diagram*", "*Component Diagram*", "*Deployment Diagram*", "*Object Diagram*" and "*Package Diagram*".

Behavioral, on the other hand, emphasizes the jobs or the operations that the system performs. It contains: "*Activity Diagram*", "*Use case Diagram*", and "*State Machine Diagram*".

Interaction more specifically describes the flow and order of data in the system. It contains: "*Sequence Diagram*", "*Communication Diagram*", "*Interactive Overview Diagram*" and "*Timing Diagram*".

Figure 1 shows the different views of UML listing all diagrams under each view.

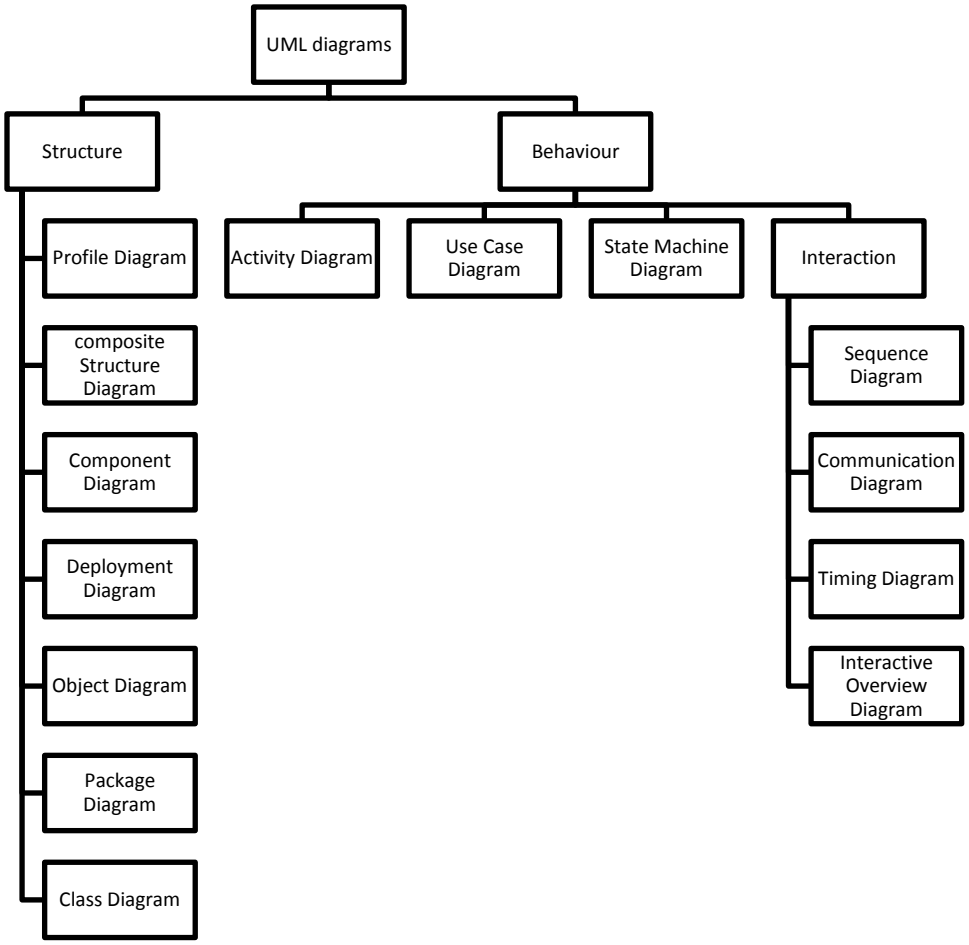

Figure 1: UML Views and Diagrams [1]

As laid out in our research objective chapter (chapter 4), a significant contribution of this research is to refactor multi-view UML model and compare the results with individual UML diagram. Multi-view UML model was proposed by Misbhauddin where one diagram was selected from each view. The selected diagrams were: Class Diagram, Use case Diagram and Sequence Diagram. A short description of these diagrams along with Misbhauddin proposed model is given in the following subsections [16].

2.1.1 Class Diagram

Class Diagram is defined by the UML reference manual as "a graphic presentation of the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships" [17]. Class Diagram is the most common UML diagram used by numerous software designers. This is due to its ease, flexibility and resemblance to the software code. Its ease can be inferred from the fact that it is representing a static state of the system and it can be transferred to code quickly. Its flexibility is clear since adding or removing any component can be done without affecting the whole design. Its resemblance to software code is obvious from its representation.

Class Diagram is represented as a block with three parts: the upper part contains class name, the middle includes class data members and the lower includes class methods. Each class diagram may connect with other classes in the system using some relationships such as: association and inheritance. Figure 2 shows an example of a class diagram:

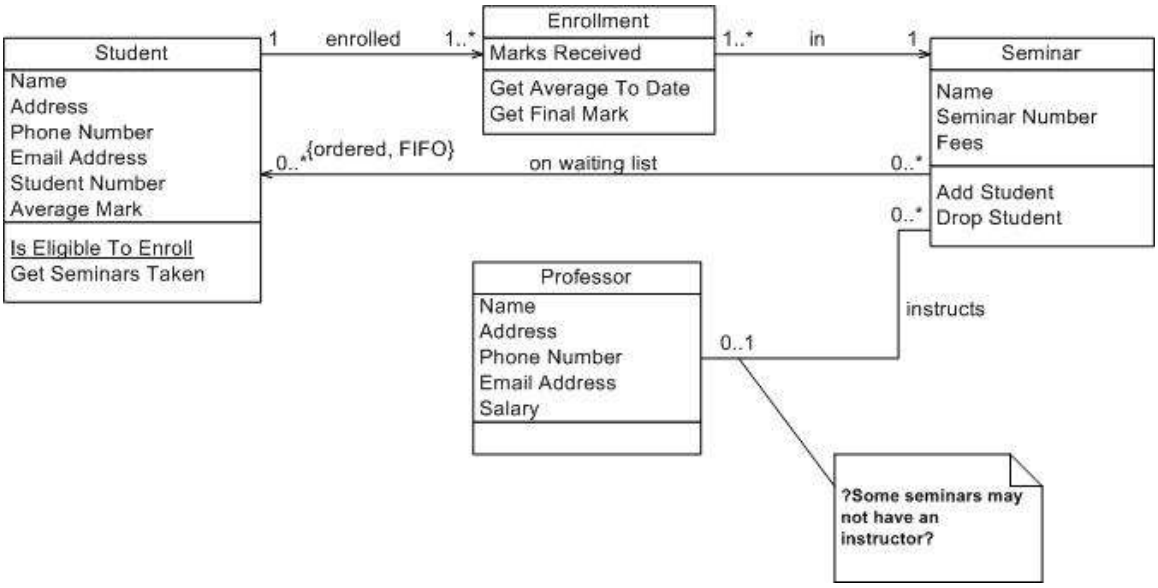

Figure 2: Example of a Class Diagram [2]

2.1.2 Use case Diagram

Use case is defined by the UML Reference Manual as “The specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or a class can perform by interacting with outside actors” [17]. Use case diagram provides a higher abstraction of the system as a whole. It denotes the players that interact with the systems either internally or externally known as actors and it specifies the major functions of the system. The actors represent users interacting with the system. The user can be a human or another subsystem.

The use case diagram is very important and used by many designers due to many reasons. First its high abstractions helps any of the stakeholders , not even the technical, to understand what the system is about and what functions it is going to achieve. Another important reason stems from the fact that it helps all stakeholders to decide on the scope of the system. This is a necessary factor to avoid feature creeping throughout the project, which is an important consideration for project managers. Lastly, the use case diagram is critical for some software development methodology such as Unified Process (UP) [18, 19]. Figure 3 shows an abstract example of use case diagram

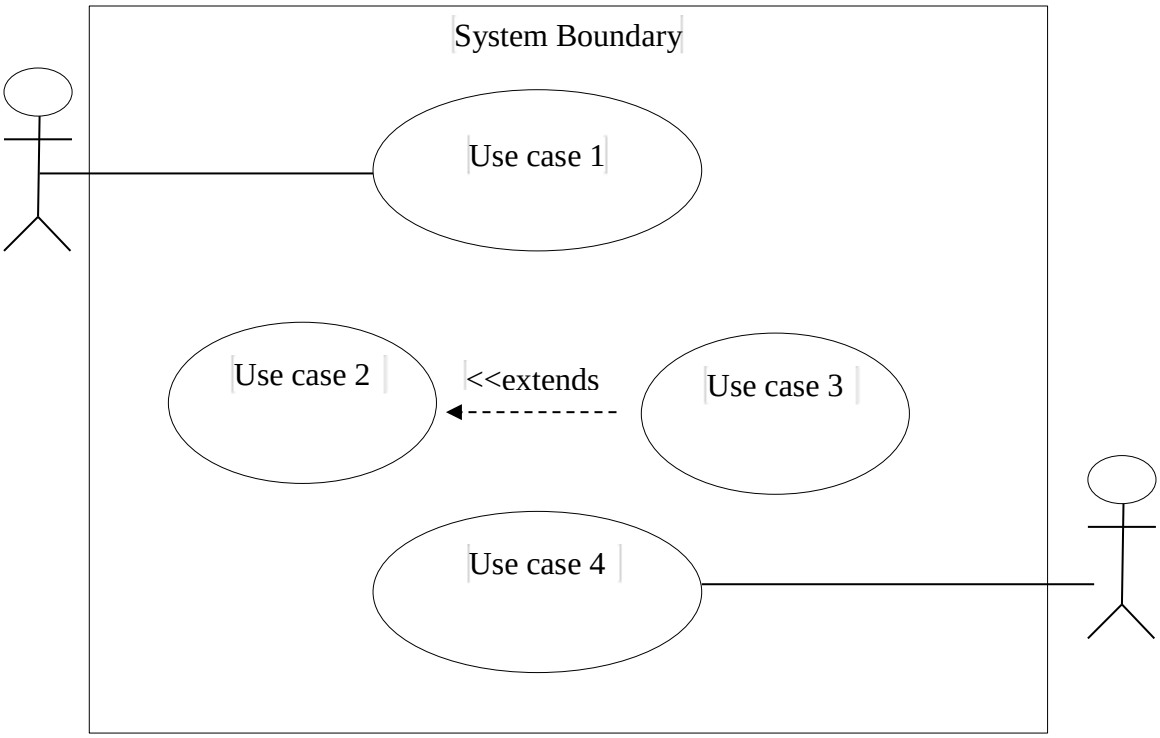

Figure 3: Abstract Example of a use case Diagram

2.1.3 Sequence Diagram

The Sequence Diagram is defined by the UML Reference Manual as "A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in an interaction and the sequence of messages exchanged" [17]. The sequence diagram is a dynamic UML diagram that shows the interaction and the order between the components of the system. The sequence diagram is about how different objects of the system interact over time via messages. It represents objects as vertical lines and messages as arrows with labels. The sequence diagram is not intended to depict complex systems due to its extensive details. Nevertheless, it is preferable for developers because it increases the level of understanding of how different objects are implemented in the system. The sequence diagram can be considered as a protocol definition of some tasks [18]. Usually, the sequence diagram should not be that large and it should correspond to one scenario only. Figure 4 shows an example of a sequence diagram for Email system adopted from [20].

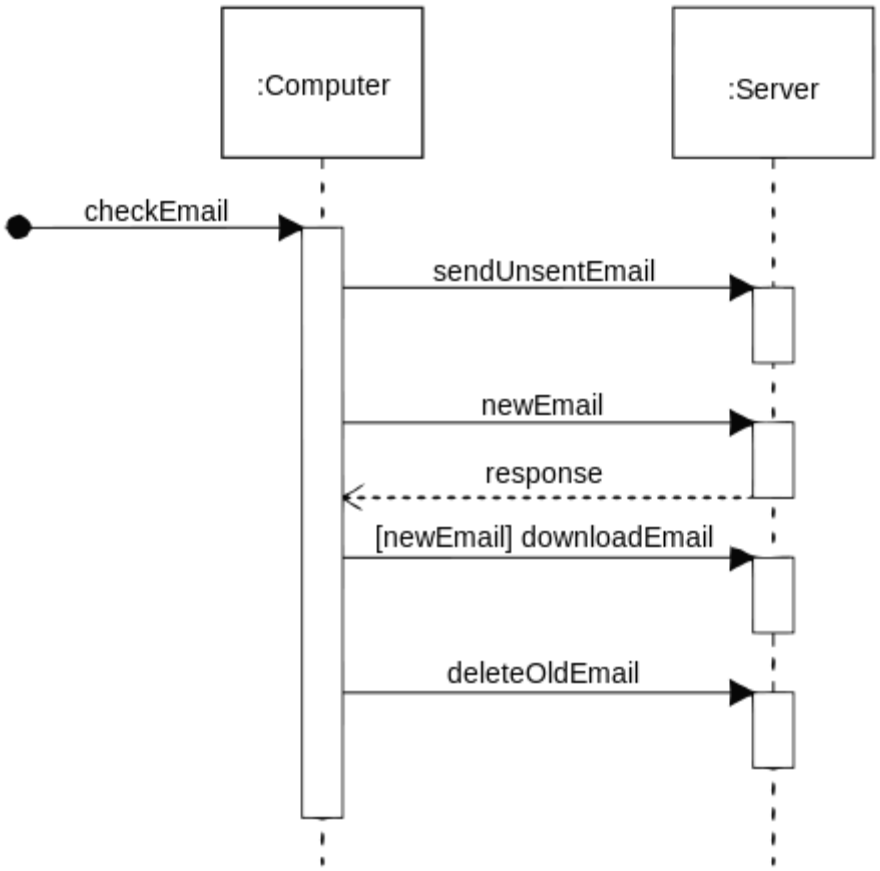

Figure 4: Example of a sequence diagram for Email system [20]

2.1.4 Multiple-view UML

Multi-view UML model is a novel model proposed by Misbhauddin as a multi-view [16]. A candidate diagram is selected from each view and a multi-view UML model is constructed from these three views namely Class Diagram from the structure view, Use case diagram from the behavior view and sequence diagram from the Interaction view.

It is worthy to note that in Misbhauddin work, the views are called: Structural, behavioral and Functional since he adopted Iivari's classification [21]. Sequence diagram belonged to behavioral view and use case diagram belonged to Functional View. However, in this research, we referenced the OMG UML [1]. Regardless of these minor changes, Misbhauudin's work can be used to represent a multi-view UML model. Figure 5 shows the multi-view UML model Metamodel adopted form [16].

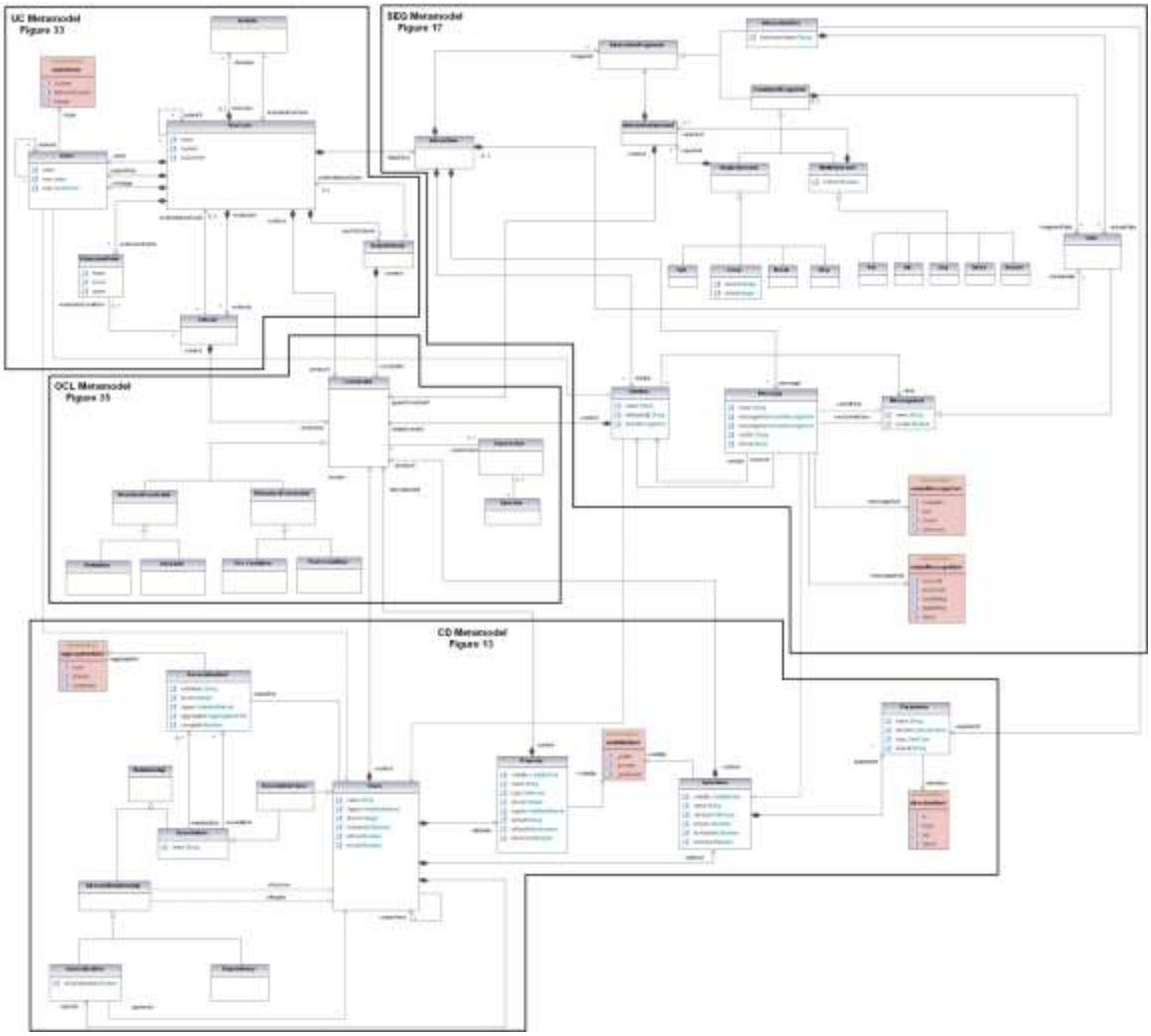

Figure 5: Multi-view UML model [16]

2.2 Refactoring

In this section, we discuss code and model refactoring.

2.2.1 Code Refactoring

Code refactoring is the most popular type of refactoring discussed in literature [4]. Code refactoring aims to improve the quality of the code without altering the external behavior. Since programmers differ in their skills and abilities to write a clean code, there is always the desire among software organizations to improve code in order to increase software quality. There are different approaches for code refactoring proposed in literature. These approaches can be categorized into three categories: rule-based, pattern-based and metric-based [22].

Recently, Artificial Intelligence (AI) refactoring was applied for automatic detection and refactoring of code smells [6]. AI techniques can be utilized for any of the above categories but applied in the literature mainly for the metric-based approach [6, 23, 24]. Below, is an example of code refactoring using C++ language [3]:

Original Code

```
int temp = average /no_of_students;
cout>>temp;
int temp = total/no_of_students;
cout>>temp;
```

Refactored Code

```
int tempAverage= average /no_of_students;
cout>>tempAverage;
int tempTotal = total/no_of_students;
cout>>tempTotal;
```

The above original code is confusing due to the multiplicity of using *temp* variables. It is not necessary for *temp* variables to be close of each other to tag it as a need-to-be refactored. Even if a variable is scattered in different files, it still causes confusion. The refactored code eliminates any sort of confusion or ambiguity and it will aid in debugging the code by giving a meaningful name for each variable.

2.2.2 Model Refactoring

As with code refactoring, design refactoring attempts to improve the design of UML models to save the maintenance cost and effort.

In comparison to code refactoring, few studies discussed UML refactoring. This can be attributed to three factors: the higher abstraction in UML models and the late maturity, adoption of UML in software industry and adoption of agile processes. Agile software development methodology implies developing software code in evolutionary manner and it doesn't emphasize much on the design.

Figure 6 and Figure 7 illustrate a simple example of UML design refactoring [3].

Original Class Diagram

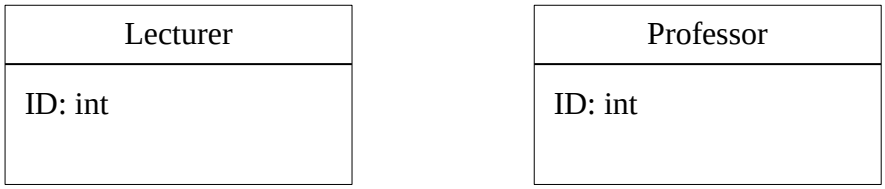

Figure 6: Non refactored UML model

Refactored Class Diagram

In the above example, we notice that "***Lecturer***" and "***Professor***" classes share the same data member "***ID***". Therefore, it would be better to create a superclass that captures similar data members of these two classes. A suitable class will be named "***Faculty***" where "***Lecturer***" and "***Professor***" classes will inherent the similar data members.

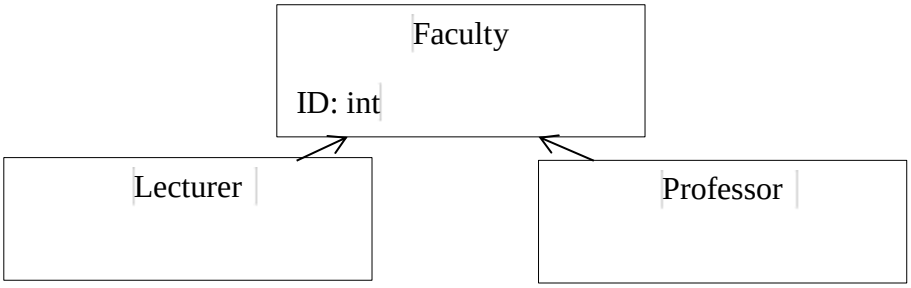

Figure 7: Refactored UML

2.3 Transformation

Refactoring is applied to software code to make it better. This is the primitive idea of refactoring. When it comes to UML model refactoring, the idea is different. Code is easier to be handled, processed and manipulated by machine. On the other hand, models are easier for human to be understood, processed and manipulated. Hence, most of the refactoring tools, approaches, process in the literature target software code. To refactor a model, it can be first transformed either to another model [25] or to a text-based description [26].

Model transformation takes several forms [27]:

- Direct manipulation of the model using a set of API provided by the model language.
- Intermediate representation such as XML.
- Language constructs [22, 28-30]

2.3.1 Model-Model Transformation

In this type of transformation, a UML model is transformed into another UML model using some languages such as Atlas Transformation Language (ATL) or Query-View-Transformation (QVT). This approach is mostly based on OCL language.

2.3.2 XML

XML Metadata Interchange (XMI) is a standard format proposed by OMG for data exchange between XML documents. Both XML and XMI are very useful in transforming UML diagrams.

Transforming the model to XML/XMI is desirable because it allows working with data and metadata, it enables to produce XML documents that can be easily interchanged and because of the availability of tools that perform the transformation. Notwithstanding, there are some drawbacks of applying this approach such as: it tends to be lengthy [31], reverse engineering is not always possible, and it requires validation since it is not formal.

2.3.3 Logical Transformation

This approach describes models using logical notations and infers some formulas or rules that assure the transformation from source to target model. Boolean logic and algebra operators are used to specify the rules and the constraints of transformation [32].

2.4 Metrics

In order to apply Artificial Intelligence (AI) techniques for UML refactoring, some quantitative metrics should be used as objective functions. These quantitative metrics will guide the AI algorithm on how to optimize the returned solution. As explained in section 2.5, most of optimization algorithms require fitness (objective) functions to optimize. Metrics is a standard measurement on which a judgment can be decided.

Metrics is very crucial for research in empirical software engineering due to its power in determining the success or failure of a proposed process, tool or a product [33]. In software engineering literature, metrics refer to the field of measurement or to the entity under measurement [33]. Fenton and Pfleeger [34] indicated that software Metrics is applied at a wide spectrum of software engineering activities.

Metrics are either objective following some criteria or subjective based on some qualities. An example of objective metrics is line of code (LOC) or the number of actors in use case diagram. Subjective metrics can produce different values based on different judges. Readability for example can be rated differently by different judges.

If a metric is obtained directly from software artifact, it is called direct measure. For example, number of methods can be extracted from the source code. Otherwise, if it is calculated based on other metrics, then it is called indirect [33]. For example, lines of codes and number of functions metrics can determine software complexity. So complexity is classified as an indirect metric.

In that regard, software metrics can indicate quality attributes of the software either internally or externally. Internal quality attributes imply that the attribute can be measured without executing the code; for example, Lines of Code (LOC) metric. On the contrary, external quality attributes cannot be measured unless the code is executed such as: number of bugs or defects in the program.

Many metrics were proposed to measure software code quality. These metrics may not be applicable to measure models quality [35, 36]. Nevertheless, they are used by many researchers [37] in the literature such as Chidamber and Kemerer (CK) Metrics [38]. In line with that, other researchers continue extending the work by investigating and empirically proposing new metrics suitable to design models including UML such as Quality Model for Object Oriented Design (QMOOD) [39]. Another approach is to realize some qualities attributes such as: reusability, readability...etc. as refactoring metrics [5].

2.4.1 CK Metrics

Chidamber and Kemerer [38] proposed six metrics to measure object-oriented systems. These metrics are: *Weighted Methods per Class (WMC), Coupling Between Object Classes (CBO), Response for a Class (RFC), Depth of Inheritance Tree (DIT), Number of Children (NOC), and Lack of Cohesion of Methods (LCOM).*

2.4.2 QMOOD

Another metrics suite to measure object-oriented systems is QMOOD [39]. QMOOD suite contains the following metrics: *Design Size in Classes (DSC), Number of Hierarchies (NOH), Average Number of Ancestors (ANA), Data Access Metrics (DAM), Direct Class Coupling (DAC), Class Interface Size (CIS), Measure of Aggregation (MOA), Cohesion among Methods of Class (CAM), Measure of Functional Abstraction (MFA), and Number of Polymorphic Methods (NOP).*

2.4.3 Quality Attributes

As explained in the metrics subsection (2.4), metrics can be related to quality attributes. Some authors in the literature validate their refactoring effects using quality attributes. For example, Alshayeb [5] empirically investigated external quality attributes which are affected by refactoring. These attributes are: *Adaptability* refers to how easily a component can be modified when imported to another environment. *Maintainability* indicates the ability to of the component to satisfy three maintenance types, which are correction, perfection, adaptation. *Understandability* measures how easy for the user to understand a component. *Reusability* is the ability to implant the component anywhere in the system with little or no modification. *Testability* implies the effort that should be taken to validate the component.

2.5 Artificial Intelligence Algorithms

There are various techniques to automate refactoring process such as: search-based, graph transformation and logical rules. In the following subsections, we provide introduction to the Artificial Intelligence techniques that can be used for refactoring. These techniques engage some intelligence in their implementation. Three types of techniques are discussed: ***"Metaheuristics"***, ***"Soft Computing"*** and ***"Machine learning".*** In this research, we will evaluate the applicability of using these techniques and we will be using the most applicable ones to perform refactoring.

2.5.1 Metaheuristics

Metaheuristic is a set of algorithms that search for an optimal solutions driven by two main forces: Intensification and Diversification [40]. Their intrinsic movement is inspired from nature or biological observations such as: Evolution or Ant Colony. They have been applied extensively with great success in various domains. We are giving examples of four algorithms in this section namely: Genetic Algorithms (GA), Ant Colony Optimization (ACO), Tabu Search (TS) and Hill Climbing (HS).

Genetic Algorithm

Genetic Algorithms are a subset of a larger class of algorithms named Evolutionary algorithms where the basic theme of these algorithms is to mimic biological nature by evolving the current population in an iterative manner to produce better solutions in each subsequent generation. GA works on random initial solutions (called chromosome) that is composed of variable-size blocks. At each iteration, these blocks are either mixed or flipped using some operators such as: crossover and mutation to produce new chromosomes with higher values calculated by objective function “fitness”.

Various implementations of GA operators with many variations, extensions and enhancements have been suggested, developed and applied in the literature [41-44]. As such, GA is considered as a primary candidate of comparison for any new developed metaheuristic algorithm [45].

Cuckoo Search:

Cuckoo Search (CS) is a recent metaheuristic proposed by Yand and Dep [46] to solve various optimization problems. The idea is inspired from nature where cuckoo lays one

egg each time and disposes it at a random nest. In each iteration, the high quality nests will be able to keep its eggs to the next generation.

Ant Colony Optimization

Ant Colony Optimization is an instance of a larger class of algorithms named Swarm Intelligence. The algorithm was a result of a PhD Dissertation proposed by Dorigo in 1982 and was published later in a book [47]. The basic idea of this algorithm is to mimic ants' behaviors when searching for food and apply it on problems where the objective is to find a shorter path between two nodes. Ants travel in random paths during their search for food. Each ant leaves some pheromone that evaporates after a specific time. The most desirable road will attract more ants and it will be marked by many pheromones left over. Hence, this will be an indicator of the optimal path between nest and food source.

ACO were successfully applied in various problems [48-51]. As with other metaheuristics, various proposals of extensions, variation or adjustments have been discussed in the literature [52, 53]. Many algorithms were developed in the same way by observing animal behavior in different situations. Bee colony, bat algorithm, firefly and cuckoo search [46, 54-56] are few to mention.

Tabu Search

Tabu search was developed by Glover in 1986 in his classic paper [57]. Tabu search is a search algorithm that was created for the purpose of escaping local minima and exploring new paths during the search. Tabu search starts from a particular solution and check its neighbors. Potential neighbors that might contribute to the global optimality path are saved in a "Tabu list". This list functions as a memory to prevent the Tabu list from

reversing to a poor solution and it strengthen the search by providing more solutions to explore in a case of a local minima trap.

Two features distinguish Tabu lists from other metaheuristics: memory and non-randomization. Glover in his paper [57] argued that Tabu search strategy is not in favor of a good solution that is found by an accidental randomness of the algorithm.

Hill Climbing

Hill Climbing or greedy algorithm is the simplest algorithm for optimization problems. The algorithm works in a greedy manner by moving from one point to another if the latter is more optimized than the former according to some objective function. If all neighborhoods of a current solution are inferior, then the algorithm is terminated. This leads to a situation referred in the optimization literature as “local optima”.

Hill Climbing is not intended to find global optimality in a solution space since it is not equipped to escape the local optima [58]. However, its strength in exploitation (local search) enables it to be a primer candidate in a hybrid metaheuristic algorithm.

2.5.2 Soft Computing

Soft Computing is intended mainly for problems that do not require an exact solution. Instead, any solution ranges within the threshold is acceptable. These techniques are very helpful when there is uncertainty involved in input collection, data measurement or output calculation. Two major algorithms are discussed in the literature review: Fuzzy Logic and Neural Network.

Neural Network

Neural Network is a computation algorithm resembling the brain. It is a schematic graph of a set of nodes called input, another set of nodes called output and a set of hidden unknown layers connecting both ends. Neural networks are ultimately used for training. The strength of neural networks comes from its capability to be a function approximation. Input and output nodes or neurons are connected with different values of weights by adjusting the weights. Artificial Neural Networks (ANN) is capable of minimizing the error of mapping input to output. Based on the number of neurons in each layer and the number of hidden layers between the input and the output, neural networks are believed to be capable of mapping any function theoretically.

Fuzzy Logic

Fuzzy logic was introduced by Lotfi Zadeh in 1960s [59] as an extension to the classical logic that computers can handle. Computers essentially are able to handle two truth-values for any statement: True and False only. Since most of the problems in the nature cannot be treated with such a simple logic, fuzzy logic along with fuzzy set help computer machines to address problems with various degree of truth.

Fuzzy logic has been applied extensively in various computer problems due to its ability to handle noise, outliers and uncertainty [60, 61]. The proof of fuzzy logic to be a universal approximator motivated many researchers to apply it to various problems since theoretically it will converge to an optimal solution [62]. Moreover, the inherent mechanism of fuzzy logic that allows it to handle different degree of truth makes it a preferred candidate in modeling subjective measurements.

2.5.3 Machine Learning

Machine learning is a set of techniques that enables the computer to learn. They are intended to infer some information or knowledge from raw data. They are mainly composed of three categories: Clustering, Classification and Association. In the literature review, we only discuss clustering algorithms as they have been already applied in refactoring.

Clustering

Clustering is considered as one type of data mining and machine learning process. Its popularity is attributed with the increase of interest in internet and the huge demand for analyzing large internet data.

Clustering is a process that collects data objects that are similar to each other in one group and dissimilar in other groups [63]. The algorithm has no idea on guidance on the objects and iteratively tries to collect data from these objects and divide them into various groups called clusters. Thus, though they are numerous algorithms in the literature, most of them are based on similarity measures. Unlike metaheuristics, where problem types play the major role in determining the difficulty of the problem to be solved by a certain metaheuristic, in clustering, data types contribute the most to clustering algorithms. This observation is obvious since metaheuristics collect information from the problem or solution space to guide the search, where in clustering techniques the objects data (attributes, features, types ...etc.) guide the clustering algorithm.

2.5.4 Multi-Objective optimization (MOP)

Multi-Objective or multi-criteria optimization [64] involves many objective functions to optimize and the optimum is not a single point but it is a set of solutions named Pareto [65]. It is possible to convert MOP problems into a single objective optimization using some techniques such as: weighted sum and utility method [66]. The difference between these two methods is that utility methods consider uncertainty while weighted method is deterministic. Another solution is to select one objective as a main objective and rewrite the others as constraints.

2.6 Refactoring and Optimization

Refactoring can be considered as an optimization search problem as follows:

- Given a search space (S), then any point (x) represents one model instance.
- The refactored model in (S) is η where for any x in (S): $\eta_\alpha \geq x_\alpha$ according to some metrics α.
- The objective function f(x) is a mapping from $x \rightarrow \bar{x}$ in (S)At each evaluation of f(x),

$$X_i = \begin{cases} X_i = x_i & \text{if } f(x_i) = f(x_i) \\ X_i = x_j & \text{if } f(x_j) < f(xi) \text{ and } i \neq j \end{cases}$$

- At any time, if x_i is evaluated by $f(X) = \eta$, then x_i is the optimum refactored model, otherwise x_i is just a refactored model.

- From the above points, refactoring (R) as a search space is represented by the tuple R < S, x, f(x)>

CHAPTER 3

LITERATURE REVIEW

3.1 Refactoring

Refactoring is defined by Opdyke, as "program restructuring transformation that supports the design, evolution, and reuse of object-oriented application framework" [67]. Fowler, who wrote the classic reference in code refactoring, defined it as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [3].

There are several advantages of software refactoring in spite of its cost in terms of time and money [3] such as: Improving the design of software, making it easier to understand, helping you find bugs in your program faster.

Mens and Tourwe illustrated refactoring process as follows [4]:

- Identify where the software should be refactored.
- Determine which refactoring(s) should be applied to the identified places.
- Guarantee that the applied refactoring preserves behavior.
- Apply the refactoring.
- Assess the effect of the refactoring on quality characteristics of the software or the process.

- Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, test, etc.).

Model Refactoring is performed to satisfy specific design qualities. There are some reasons that urge decision makers to work on model refactoring [68]: to meet design goals, to address deficiencies uncovered by design analyses and to explore alternative designs. The first to start refactoring UML diagrams was Sunye et al. [12].

3.1.1 Refactoring Metrics

. To be able to validate the impact of refactoring on the model quality, metrics are a popular approach. Mens asserted the need of continuous research in measuring the impacts of refactoring on software quality by using a set of right metrics [69]. These set of metrics would aid in spotting the areas where refactoring should be applied [70]. Performing refactoring using some metrics was first originated by Simon et al. [71].

Metrics received wide attention from researchers to propose new metrics, to argue about the validity of metrics or to introduce formal definition. ReiBing [72] proposed an object-oriented design metric named ODEM which is based on UML meta-model. Some new metrics introduced specifically to UML models targeting the model, class, messages and use case were proposed by Kim and Boldyreff [35]. Fenton [73] argued about the need to build some scientific basis for software metrics. McQuillan and Power [37] specified a formal definition for the popular Object-Oriented (OO) C&K metrics. Rudiger and Lowe [74] proposed a description framework for defining software metrics composed of : Description, Scope, View, Definition, Scale and Domain.

It is important to be able to measure metrics from UML diagrams and source code accurately since: it is cost-effective, to be able to monitor the deviation of the code from the planned design and for evaluation purposes from design-to-code and code-to-design [36]. As conjectured by Mcquillan and Power, code metrics can be used for measuring UML design qualities as well [37]

3.1.2 Model Transformation

Czarnecki and Helsen [31] classified different types of model transformations. They acknowledged the complexity of graph-based transformation and the impractically in direct manipulation of the model.

Dominguez et al. [75] presented an interesting comparison between various transformation methods from UML to XML. The comparison is based on a framework composed of nine criteria namely: traceability, Incrementally, Metamodel approach, category, number of metamodel, kind of transformation, paradigm and tool. They found that some studies use Model-to-Text-to-Text or even Model-to-Model-to-Text. Their work was focused on UML class diagram only.

Mens and Gorp [76] proposed a new taxonomy of model transformation approaches. Their work is considered a premier reference on various properties and issues of transformations. Massoni et al. [32] applied refactoring on class diagrams with semantic-preservation condition. The refactoring was based on Alloy, a formal modeling language.

Misbahuddin and Alshayeb [77] compared the different transformation approaches which are: Graph-Based, Logic-Based, Direct Manipulation, Language Specific and Text based.

Text based was the only candidate that satisfied the following criteria: Easiness, Automation and Coverage where it is not very complex to be used for refactoring, and it is fully automated where user intervention is not required (suitable for Artificial Intelligence refactoring) and it covers the three UML views.

3.2 Code and Model Smells

A code bad smell indicates that even the program is running; there is a potential that problems might lead to some defects. A bad smell is the core idea that refactoring process is built on. Refactoring intends to improve the software to save maintenance cost and effort. Identifying the bad smells that might contribute to some errors in the software is very essential for refactoring process. Many studies are dedicated to identifying bad smells.

Nevertheless, in the last two years a new direction named as refactoring opportunities starts to attract researchers' attention [78]. In refactoring opportunity identification, the software code is searched for places where a certain refactoring operation can be performed. Some of the common bad smells with a brief description are provided in appendix B.

In lieu of that, model refactoring is concerned about model smells. Due to the various diagrams in UML, each model has different smells. Class Diagram is similar in representation to the class code and hence they share many of the smells. However, for other UML diagrams, it is not clear how to determine that a model may exhibit some deficiencies.

Use case diagram smells can be identified using anti-patterns. An AntiPattern is "a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences" [79].

Thus from the above discussions, model smells refer to deficiency in the model that might cause a problem or it might refer to anti-pattern as well.

3.2.1 Refactoring Operations

Refactoring Operations, also known as refactoring activities, refer to the operations applied on the original code to eliminate bad smells. There are over 70 refactoring operations in the literature suggested by many authors [7, 80, 81]; however, only few of them are studied in the literature or implemented in practice [78].

Most of the refactoring activities are presented in Fowler Catalogue [3, 80, 82]. However, these operations are addressing ill-code structure. For model design, we are not aware of any specific catalogue that addresses all ill-structured UML diagrams. However, El-Attar and Miller [30] proposed a comprehensive list of refactoring activities addressing ill-structured use case diagram; they referred to this list as anti-patterns [30]. In addition, some of code refactoring activities can be applied and adapted to model refactoring. An example of code refactoring activity that can be adopted for refactoring class diagram is *"Extract Class"*. Ghannem et al. [13] ran their experiment of model refactoring on twelve of Fowler code-related refactoring activities Catalogue.

Al-Dallal [78] reviewed the literature thoroughly in his Systematic Literature Review (SLR) and found that metric-based approach is the most popular approach used in the

literature for identifying refactoring activity. Thus, we are going to use this approach in our research. In addition, Al-Dallal also found that a considerable number of refactoring activities were not applied in any research [78]. More than 50 of the refactoring activities provided in Fowler catalogue were not addressed by any researcher. Hence, he concluded that there is a need for more studies that investigate: Rename field, Rename Method, Inline Temp and Add parameters operations due to their frequent use in the industry.

We provided an example of refactoring activity in the background and some of model smells and anti-patterns are presented in Appendix B.

3.3 Refactored Diagrams:

This section provides a description of the three diagrams that we are going to use for refactoring.

3.3.1 Use case Diagram

The UML use case diagram is mainly used for requirement elicitation. It provides a high abstraction of the system as a whole and denotes the players that interact with the systems, either internally or externally. These players are known as actors. It shows the major features of the system and how these features are shared among different actors. Formally, the UML Reference Manual defines the use case diagram as: “the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or a class can perform by interacting with outside actors” [17]. The actors represent users interacting with the system and they can be either humans or other subsystems.

The use case model is composed of diagrams and descriptions accompany these diagrams. When refactoring use case models, some researchers focus on the design level

which is the diagrams [25], whereas others address the requirement level which is the description [83, 84].

Use case diagram refactoring is done either on episodes [85] cascaded [86] or is anti-pattern based [25]. Yu et al. [85] worked on an episodes model on refactoring use cases. Each episode corresponds to one rule applied to a use case with a total of ten episodes. Cascaded refactoring was proposed by Xu and Butler [86] to extend a refactoring step to three models namely: the feature model, use case model and architecture model. El-Attar and Miller presented anti-patterns as a mechanism for use case modeling [87], applying anti-patterns to improve the quality of use case models. Later, Khan and El-Attar [25] used anti-patterns for refactoring use cases. Their work presented several examples of anti-patterns in use cases that can be refactored by model transformation. A full list of anti-patterns that can be applied to use cases is presented in [88].

3.3.2 Sequence Diagram

A sequence diagram is defined by the UML Reference Manual as “a diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in an interaction and the sequence of messages exchanged” [17]. A sequence diagram is a dynamic UML diagram that shows the interaction between the components of the system. A sequence diagram shows how different objects of the system interact over time via messages. It represents objects as vertical lines and messages as arrows with labels. A sequence diagram is not intended to depict complex systems due to their extensive detail. Nevertheless, it is useful for developers because it increases the level of understanding of how different objects are implemented in the

system. A sequence diagram can be considered as a protocol definition of certain tasks [18]. Usually, a sequence diagram is not large and it should correspond to one scenario only.

Several studies on refactoring UML models have been proposed. Mens et al. [89] acknowledged the usefulness of performing refactoring on higher abstract levels of a software system, such as design levels. Sunye et al. [12] started the research in the UML refactoring domain via their well-known article, Refactoring UML Models. They illustrated refactoring rules on two popular UML diagrams: class and statechart diagrams. As acknowledged in their paper [12], finding refactorings in UML diagrams is not straightforward and further research is required.

Misbhauddin and Alshayeb [77] compared different approaches used in the literature for refactoring UML diagrams. They constructed a criteria-based framework for comparison. The selected approaches are: graph-based, logic- based, direct manipulation, language specific and text-based approach. The following criteria were chosen to compare these approaches: object-oriented concepts, formality, ease of use, conciseness, artifact coverage, expressiveness, granularity, automation, portability and rule handling. Their article provides a holistic view of the merits and drawbacks of each approach for any researchers interested in refactoring UML diagrams.

Misbhauddin and Alshayeb [90] searched the literature on UML refactoring using a systematic literature review and found that only 16% of papers dedicated to UML refactoring discuss sequence diagram refactoring.

Al Dallal [91], in his systematic literature review, illustrated that the most common approach to identify refactoring actions is to utilize quality metrics, with around 32% of all papers applying refactoring operations. Al Dallal also indicated that clustering techniques for refactoring were utilized by around 23% of all surveyed papers. The clustering techniques were mostly based on the similarity between two methods or between a method and attributes. In addition, Al Dallal asserted that cohesion and coupling metrics are the most commonly used in the existing studies to apply quality metrics to evaluate the refactoring process.

Maneerat and Muenchaisri [92] proposed machine learning techniques for bad smells detection of UML diagrams. They applied seven different machine learning algorithms to detect various bad smells such as: lazy class, message chains, middle man, etc. However, they restricted their research to class diagrams only.

Fourati et al. [93] applied quality metrics in order to detect anti-patterns in some UML diagrams, including sequence diagrams. Their work revealed that the cohesion metric, in line with other metrics, can aid in detecting some abnormality in the sequence diagram. They illustrated with examples how quality metrics can unleash four common anti-patterns namely: Blob, Lava Flow, Functional Decomposition and Poltergeists. However, unlike our research, they did not apply any search-based algorithms to detect abnormalities and refactoring to refactor them.

Alkhalid et al. [7, 94, 95]applied clustering techniques at different software levels. They showed that clustering can improve the cohesion and coupling metrics of software code if similarity distance is considered. In their papers, they applied four different clustering

algorithms on different open source projects with a fixed and variable number of software entities under study, which are package, class and function. Their research shows that clustering can be very promising in providing refactoring decisions to the user which clearly reflects on the quality. In [7] , they applied clustering to a software package and highlighted that these package classes can be considered the entities while the methods are considered the features, thus an entity-feature matrix can be constructed to guide the clustering algorithm. In [94], they applied clustering to software classes and assume that the function should be moved to a certain class based on the number of attributes it accesses. Therefore, the methods are the entities and the class variables are the features. Similarly in [95], they constructed an entity-feature matrix by considering the function statements as entities and their attributes as the features.

Ghannem et al. [13] used an interactive genetic algorithm that prompts and interacts with the designer to help him to refactor a class diagram. Their paper targets a learning-based algorithm where the algorithm learns from a base of examples in order to generate refactoring decisions to the user. However, their data is converted from code to UML diagrams using a tool. Their articulation of the interactive genetic algorithm steps to model refactoring can help researchers to understand how these heuristic algorithms can be applied to refactor UML models.

In addition to these approaches, representing the refactoring problem as a multi-objective method has been discussed and implemented for software code, model refactoring and maintainability [96-99].

The above approaches suffer from drawbacks, which motivated us to consider the utility of hybridization. Evolutionary computing methods are known to be computing-intensive due to the population size and the number of generations required to converge. In addition, there are many parameters such as crossover operation, mutation operation, and selection operation etc. that should be set correctly in order for the algorithm to produce promising results.

Clustering is very effective for optimizing cohesion and coupling simultaneously but it is a computation-intensive algorithm for large data. Search-based algorithms can work only on one objective, for more than one objective, such as coupling and cohesion considered in this paper; a multi-objective version should be considered which adds more to the computational demand. Thus, combining clustering with a simple search-based algorithm such as SA could produce good results. SA has two parameters that can be easily set. In [6], only one parameter was shown to have a great impact which is the cooling factor. Thus, it motivated us to apply this hybridization for a sequence diagram refactoring.

Table 1 : Summary of related works in refactoring sequence diagram

Author and Years	Methodology Approach	Artificial Intelligence Algorithm	Detection / Refactoring	Refactoring Level	Evaluation Method
[7, 94, 95]	Deterministic	Clustering	Code detection and refactoring	Function Class Package	Coupling and cohesion metric
[13]	Search-Based	Interactive Genetic Algorithm	Class model detection and refactoring	Method & Class	Recall and Precision
[92]	Deterministic	-	Class model detection	Method & Class	Accuracy, Specificity, sensitivity ,..etc.
[96]	Search-Based	Genetic Programming and Genetic Algorithm	Code detection and refactoring	Method & Class	Recall and Precision
[97]	Search-Based	Non-Dominated Sorting Genetic Algorithm –II.	Code detection and refactoring	Method & Class	Recall and Precision
[100]	Search-Based	Genetic Programming	Code detection and refactoring	Method & Class	Recall and Precision

[101]	Hybridization of Deterministic and Search-Based	Neural Network and Genetic Algorithm	Code detection and refactoring	Method & Class	Recall, Precision and Refactoring Efficiency
The proposed approach	Hybridization of Deterministic and Search-Based	Kmeans and SA	Sequence diagram detection and refactoring	Method & Class	Cohesion, Coupling, Recall and Precision

3.3.3 Class Diagram

Class Diagram is defined by the UML reference manual as "a graphic presentation of the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships" [17]. Class Diagram is the most common UML diagram used by numerous software designers [22]. This might be attributed to its ease, flexibility and resemblance to the software code. Its ease can be inferred from the fact that it represents a static state of the system and it can be transferred to code quickly. Its flexibility is clear since adding or removing any component can be done without affecting the whole design. Its resemblance to software code is obvious from its representation.

Software quality is gaining more interest among researchers and practitioners in the past decades [102-104]. This is attributed to the cost and effort spent on maintenance phase by developers [105]. To avoid that, software firms rely mostly on extensive testing and experimentations of the software [106]. However, both of these approaches are costly in term of time and money. Subsequently, many authors tried different frameworks and algorithms to detect the number of defects in software [107-110]. Since code is the main artifact of any software, a substantial effort has been done in literature to detect the defects in software code [107-110]. There are a large number of papers devoted to the

detection of class diagram's faultiness. Most of these papers target the defects of class diagrams in a code form [111-113].

Detecting bad smells at the design level was an interest to researchers in the past few years [93, 100, 114]. The benefit of detecting defects early in the design level includes saving of cost, time and effort. Detecting bad smells in class diagrams have been recently investigated by researchers [100, 115-117].

Several machine learning algorithms have been implemented and utilized to detect class defects to guide these algorithms, different sets of metrics have been used by researchers in order to obtain more accurate results [113, 118-120].

Few papers have been devoted to the use of machine learning algorithms to detect bad smells guided by software metrics [100, 117, 121]. It is understood that bad smells are just symptoms or signs that alert the designer to the probable defects but they are not defects themselves. Thus, it explains the variation of publications in these two similar research lines.

Software defects are usually detected after running the program. Smells can be detected and detected earlier. This can help programmers to estimate the number of possible defects in their software [122] and sequentially can plan testing strategy earlier. Machine learning techniques have been used in software defects as mentioned in the literature review section and that motivated us to use them for bad smell detection.

There are several algorithms that have been applied and compared in the literature to detect the class defects or smells. Gao et al.[123] considered Naïve Bayes (NB), MLP,

logistic Regression (LR), k Nearest Neighbor (KNN) and SVM to detect defects. They concluded that NB, MLP and LR obtained better results than SVM and KNN. Gondra [124] applied Artificial Neural Network and SVM for detection of software fault-proneness and found that SVM performed better than the simple ANN. Kanmani et al. [109] investigated Back propagation Neural Network (BPN) and Probabilistic Neural Network (PNN) and found that PNN is better in detecting faults in OO modulus (C++ classes). Rathore and Kumar [113] applied GEP to software defects. However, no comparison was performed with other algorithms. Maiga et al. [125] applied SVM for anti-pattern detection and found that SVM can obtain a higher accuracy. However, no comparison with other machine learning algorithms was experimented.

Fontana et al.[126] Investigated a large number of case studies in order to detect class smells. Their research considered four smells (Data class, Large Class, Feature Envy and Long Method). They experimented with a large set of different variation of machine learning algorithms found in WEKA tool. These algorithms are: J48, JRip, Random Forest, Naïve Bayes, and SVM with different variations. They acknowledged the usefulness of applying machine learning algorithm to detect bad smells with high accuracy. They found that SVM is approaching the worst performance among other algorithms. Elish and Elish [127] concluded that SVM is better or comparable to other machine learning algorithms such as: RF, NB, LR, KNN, MLP, Radial Basis Function (RBF), Bayesian Belief Network (BBN) and Decision Tree (DT) in achieving high accuracy.

Khoshgoftaar [128] stated that working with a smaller set of metrics such as 8 is appealing than working with a large set of metrics. Fontana et al. [126] applied their

algorithms on a large number of metrics and custom metrics at the project, package, class and method metrics. Since their datasets were huge and heterogeneous, such a large number of metrics may be justified. Elish and Elish [127] utilize 21 McCabe's and Halsted metrics. Laradji et al.[110] approach was guided by a 30 metrics in average for each dataset including McCabe's and Halsted metrics. Ghotra et al. [129] in their recent paper used the PROMISE and NASA metrics. Giger et al. [108] applied 15 metrics on the method level to detect method-level bugs. However, due to this huge number of metrics exist in the previous datasets, feature selection was performed to reduce the number of applied metrics.

Maiga et al. investigated four anti-patterns: The Blob, functional decomposition, spaghetti code and Swiss army knife. Fontana et al. [126] addressed data class, large class , feature envy and long method. Li and Shatnawi [119] empirically investigated the link between class smells and class error probability. The subjects of their study were: Data Class, God Class, Refused Bequest, Shotgun Surgery and feature Envy. Similarly, Dhillon and Sidhu [130] conducted similar experiments adding Long Message Chain and Interface Segregation Principle Violation. Both of the aforementioned papers concluded that there is an association between bad smells and class errors.

Ferenc [131] used machine learning algorithm to automatically detect software bugs. He used Bayes Network, Naïve, Bayes, Logistic Regression, Voted Perceptron, Decision Tree, Conjunctive Rule, J48 and Best-First Dec. Tree for training. He used a 10-cross validation approach and selected the Best-First Dec. Tree for validation. He used recall and precision measure. He acknowledged the performance of all algorithms in training.

Since he settled to use one algorithm only, he didn't make any comparison in the variation of algorithm's performance during training and validation phases.

As we can see from the previous approaches, many authors attempted to study the effect of bad smells on the class diagram. In lieu with that, some authors tried to examine the relation of the bad smell and code faults which subsequently affect the maintenance of the program. Hall et al. [132] conducted a study to investigate the relationship between code smells and presence of faults in three open source software. They selected Data Clumps, Switch Statement, Speculative Generality, Message Chains, and Middle Man as class smells candidate. They found that class smells do have significant indicators of faulty software. However, they reported that not all class smells have the same degree of severance.

Yamashit and Moonen [133] found that there are class smells and maintenance problems are associated. They conducted their empirical study using Data Class, God Method, God class, Shotgun Surgery, Feature Envy, data Clump and others. In another study by Ymashit & Counsell [134], they asserted that code smells can work as indicators for the need of maintenance. Thus, code smells are helpful in pointing out the need of maintenance, though they are influenced greatly by the system size.

We can see the importance of applying machine learning techniques that can obtain high accuracy of detection of bad smells. Since detecting as much as bad smells can reduce the number of bugs found in the software.

In this research, we used three machine learning algorithms to detect class smells. These algorithms are: MLP, GEP and SVM.

3.3.4 Multiple-View Diagram

Multi-view UML model is a novel model proposed by Misbhauddin as a multi-view [16]. A candidate diagram is selected from each view and a multi-view UML model is constructed from these three views namely Class Diagram from the structure view, Use case diagram from the behavior view and sequence diagram from the Interaction view. It is worthy to note that in Misbhauddin's work, the views are called: Structural, behavioral and Functional since he adopted Iivari's classification [21]. Sequence diagram belonged to behavioral view and use case diagram belonged to Functional View. However, in this research, we referenced the OMG UML [1]. Regardless of these minor changes, Misbhauudin's work can be used to represent a multi-view UML model.

Each UML diagram has a different set of smells or anti-patterns that alerts the designer to a potential degradation of the design quality. However, some smells are not that obvious and they cannot be detected by focusing on one diagram. As such, a manual inspection by experts looking at different diagrams belonging to different views can spot some of these smells. Thus, an integrated UML view that combines three diagrams representing different views was proposed by Misbhauddin. In his work, he asserted that an integrating UML diagram will expose some smells that was not be able to be detected from a single diagram alone.

There are some papers discussed the integrating or the combining of different models into one model. Sturm [135] proposed an integration model between domain and application

levels. A domain level is composed from features and functional model while application level is all the models instantiated from the domain model. He used resource tracking system as a case study. The purpose of his approach is to provide the right views of a system and present it to the designer in accordance with the related domain. Thus, a mapping should be established between the domain level and the application level.

Franceand et al. [136] proposed a framework for evolving object oriented software named MVSE that can represent different views of the system. They specified the relationships and dependencies between different views of UML in order to facilitate model transformations. One of the facets of model transformations that their approach is targeting is model refactoring.

Gomez et al. [137] proposed a multi-view model based on UML and two of its profiles: SYSML and MARTE. They added some stereotypes to the integrated multi-view model. Basically, their multi-view model is composed of different views where each view describes a component with all of its properties. In their approach, they keep maintaining the consistency between the different views. This is a vital issue that authors should look for when implementing multi-view approach.

Lopez-Herrojen and Egyed [138], applied safe composition approach as validating way of checking the consistency between different models in multi-view models such UM. El-Miloudi and Eettouhami [139] addressed the consistency checking of state machine diagrams in a multi-view modeling environment. However, they applied a formal semantic rules for consistency checking based on a Z specification. Their work checks the consistency between state machine diagram and class diagram and the consistency

between state class diagram and sequence diagram. They stated that their work is one of the first attempts to use formal specifications of Z notations to state machine diagram.

3.3.5 A summary of the previous work issues

The articles mentioned in this chapter suffer from some issues that we would like to address in our research:

- Most of them are targeting code refactoring [6, 23, 140, 141], while our focus is towards refactoring UML models.
- Either they adapt code-related metrics or they apply only one or two metrics [6, 141, 142], while in our research, we are going to use UML model related metrics.
- None of them is targeting a multi-view UML diagram.
- None of them is comparing multi-view of UML with a single view, while in our research; both individual UML and multi-view UML are refactored and compared using the same research settings.
- We are going to use some AI algorithms that have not applied before in model refactoring such as LAHC.

CHAPTER 4

RESEARCH OBJECTIVE AND METHODOLOGY

4.1 Research Objectives

The main objective of this research is to: "refactor UML models auomatically using a set of Artificial Intelligence techniques, that are guided by software metrics and refactoring operations, to to restucture the design to improve quality". To achieve the main objective, the following sub-objectives are proposed:

- Evaluate various AI algorithms such as: HC, LAHC and SA that can be used to refactor use case diagram?
- Evaluate various AI algorithms such as SA, a hyberdized k-means and SA and hyberdized k-means and HC that can be used to refactor sequence diagram?
- Evaluate various AI algorithms such as: Multi-Layer Perceptron, Genetic Expression and Support Vector Machine that can be used to refactor class diagram?
- Evaluate various AI algorithms such as: HC, LAHC and SA that can be used to refactor multiple-view diagram?

4.2 Research Methodology

Our Methodology is based on the following phases:

1. *Refactoring set-up:*

This step involves five components: transformation, refactoring regions, operations, algorithms and quality metrics. Transformation is necessary to convert UML diagrams

into a middle representation in order for the algorithm to run. Refactoring regions are the anti-patterns or the bad smells of UML diagrams. Some examples are provided in Appendix B. Refactoring operations are the techniques used to restructure a bad smell. AI algorithms and quality metrics are presented in the background section.

2. *Automation:*

This involves the implementation of AI algorithms, tuning the parameters and validating the results using quality metrics.

3. *Empirical Analysis:*

An empirical study is conducted based on free or commercial real-world UML models, in order to extend the validity of the results obtained in the automation steps.

We are going to use different quality metrics to evaluate the refactored diagram. For multi-view UML model, we may propose new metrics if we find that the existing metrics are not suitable to measure the multi-view UML model quality. Figure 8 summarizes our research cycle:

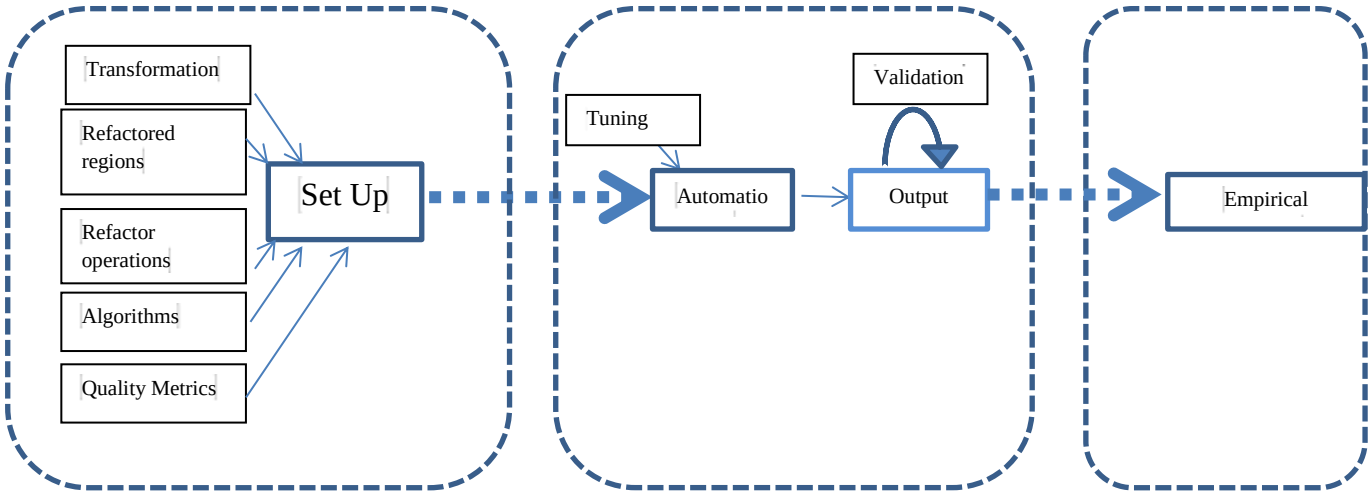

Figure 8: Research Process

4.3 Proposed solution

This part details the components of our solution strategy. The solution is similar for each diagram to ensure consistency. Each diagram is accompanied by a short description of the whole process.

4.3.1 Individual Diagram Refactoring Process

In this part, we describe with sufficient details all the processes, operations and tasks required to produce our results. The description here can be used as a reference for all four diagrams, since the flow chart of the four diagrams is similar.

- The first phase involves preparing the data, which in this part, is the structural diagram. As illustrated in the data collection subsection, the data will be either: senior students' projects, published data or commercial models. There is the possibility that some data is not clean and needs some preprocessing.

- Transforming the structural diagram to an intermediate representation of the model. This is to facilitate implementing the algorithms.

- In order to apply algorithms to the refactoring problem, we must map it to a suit of AI algorithms. Different techniques require different mappings. For examples, features extraction of the elements and comparing the similarity distance is suitable for clustering algorithm. For genetic algorithm, the diagram elements must be encoded as bits and an objective (fitness) function must be determined. In ACO, the elements are encoded as graph nodes and the optimal path would be determined by the algorithm. Mapping the problem representation to be suitable to the algorithm is known to be a very critical step and it is not always straightforward. Sometimes, it is not obvious how the problem can be encoded to be understood by the algorithm.

- This is the most important phase and the core job of this research. After obtaining the encoded version of the structural diagram, three inputs applied.

- *The particular algorithm* that we implemented. The algorithm that we selected was a general template that could be applied to various problems. Hence, some adaptations, adjustments and enhancements of the algorithm have be done to adjust it to the refactoring problem.

- *The metrics*: a set of metrics were selected that measures the quality of the structural diagram. The algorithm should refactor the model to reflect an improvement in the selected metrics.

- *The refactoring operations*: the end results will be the refactoring decisions which provide the user with the options of which refactoring sequences he prefers. It is important to note that the algorithm may not produce a good result from the first time, and thus, a repetitive tuning of the algorithm parameters might be required.

- The same process is applicable for all selected UML diagrams. However, since different diagrams portray include different components than each other; step 3 and 4 will be adjusted to suit each used diagram. In addition, since the multi-view UML model is not fully studied in the literature and hence, we presume that all steps may need some adjustments, as we may find very few multi-view UML models available.

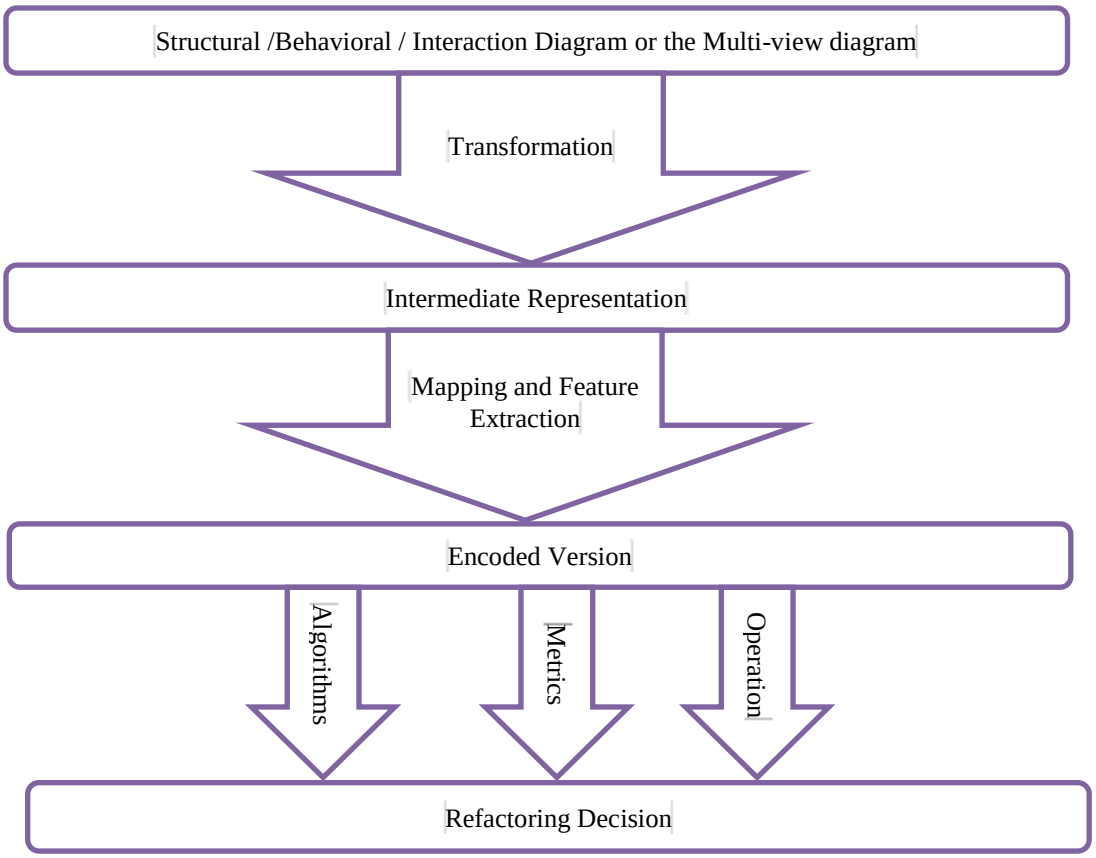

Figure 9: Proposed Solution

CHAPTER 5

USE CASE DIAGRAM REFACTORING

5.1 Case Studies

Due to the scarcity of data on real industrial UML diagrams in general and use case diagrams in particular, we adopted a published use case diagrams to show the applicability of HC, LAHC and SA for automatic refactoring. The first case study, adopted from [143] with some customization, is shown in figure 10. The second case study, adopted from [25] with some modifications, is shown in figure 11. For simplicity, we renamed the use cases to contain numbers rather than function names. In the first case study, use case 1 includes three use cases named: 1.1, 1.2 and 1.3. This resembles an anti-pattern as presented by Khan and El-Attar [25] and suggested a "Drop Function Decomposition Having Inclusion".

Figure 10 shows a use case diagram of two actors: one is generalized from the other. The first actor is connected to 8 level-1 use cases. The second actor is connected to one level-1 use case. Each use case in level 1 includes other use cases. As can be seen from the figure, the number of inclusions differs from one use case to another. Some of these use cases exhibit the specified anti-pattern, some of them do not. The search-based algorithm will search for the anti-pattern and suggests refactoring to the user. Afterwards, the algorithm will output the improvement of the used metric (CM1 & CM2).

Table 2 shows the adjacency matrix of the actors and the use cases in case study 1. As figure 10 illustrates, case study 1 has two actors and 9 use cases that communicate with these actors. For example, if there is communication between actor (1) and use case number (4), the value 1 is added in the intersection cell in table 2. If there is no communication between an actor and a use case, as in actor (2) with use case (1), we add the value 0 at the intersection cell,

Table 2: Adjacency Matrix for Actor- UC of case study 1

Actor / use case	1	2	3	4	5	6	7	8	9
1	1	1	1	1	0	1	1	1	0
2	0	0	0	0	0	0	0	0	1

Table 3 shows the adjacency matrix of relationships among use cases themselves. For example, use case number 1 (UC1) is connected with three other use cases, using the relationship “Include”, so at the intersection cell of UC1 and Include relationship, we insert the value 3. UC1 does not extend or communicate with other use cases, so the “Extend” and “Communication” columns are filled with the “0” value.

Table 3: Adjacency Matrix for UC-UC of case study 1

UC	Include	Extend	Communication
1	3	0	0
2	4	0	0
3	0	0	0
4	5	0	1
5	4	0	0
6	0	0	0
7	4	0	0
8	0	0	0
9	5	0	0

Figure 10: use case case study 1

Figure 11 shows a real world case study which contains other relations such as extend and it has one use case (1.2) that is included in two use cases (1 and 2.2). The diagram was extended by adding many instances of the anti-pattern to illustrate the applicability of the algorithm and for comparison purposes.

The adjacency matrix of communication between actor and use cases for case study 2 is shown in Table 4. The adjacency matrix of relationship communications among use cases is shown in Table 5.

Table 4: Adjacency Matrix for Actor- UC of case study 2

Actor / use case	1	2	3	4	5	6
1	1	1	0	0	0	0
2	0	0	1	1	0	0
3	0	0	0	0	1	1

Table 5: Adjacency Matrix for UC-UC of case study 2

UC	Include	Extend	communication
1	2	0	0
2	2	0	0
1.2	0	2	0
2.2	1	0	0
3	3	0	0
4	3	1	0
5	3	0	0
6	2	1	0

Figure 11: use case case study 2

5.2 Use Case Metrics

To measure the changes, which occur in the use case as a result of refactoring, we use two metrics. The first is proposed by Marchesi [144] to measure the complexity of use case diagrams. A lower value indicates better software quality since it means a less complex use case. HC, LAHC and SA algorithms will search, detect and refactor the use case diagrams based on the values calculated by this metric. The metric is calculated using the following equation [144]:

$$\mathbf{UC} = K_1\,\mathbf{UC1}^2 + \mathbf{UC3} + K_2\,[smm([\mathbf{C}]) - smm([\mathbf{E}]) \quad (1)$$

$UC1 = \sum_{i=1}^{n} Use\ case_i$

$UC2 = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}$

$UC3 = \sum_{i=1}^{m} \sum_{j=1}^{n} e_{ij}$

So $CM1 = K_1 * (\sum_{i=1}^{n} Use\ case_i)^2 + \sum_{i=1}^{m} \sum_{j=1}^{n} e_{ij} + K_2 * [\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} - \sum_{i=1}^{m} \sum_{j=1}^{n} e_{ij}]$

where,

UC1 refers to the total number of use cases

UC2 refers to the total number of communication lines between actors and use cases

UC3 refers to total number of communication lines between actors and use cases without redundancy introduced by extension or inclusion

C is a matrix whose values show the presence or absence of communication between a use case and an actor

E is a matrix whose values show the presence or absence of the "use" relationship between a use case and an actor without redundancy introduced by the inclusion relationship

K1 and K2 are constants. In our case, we set them to 0.5

The full details of the equation and the calculations can be found in the corresponding paper [144]. Table 6 shows the summary of notations used in all equations in this chapter.

The second metric is proposed by Seidl [145] based on the metric proposed by McCabe Butler [146]. It also intends to measure the complexity of the use case diagram based on the number of actors and the total number of include relationships. The value of this metric is calculated using the following equation:

$$CM2 = [1- \sum u / \sum i + \sum a] \quad (2)$$

Table 6: Summary of Notations

Relationships: i ε I: Include e ε E: Extend r ε R normal a ε A an actor u ε U a use case C is a matrix of communications E is a matrix of extend communication K is a constant U ε No of use cases A ε No of actors

5.3 Fitness Function

In order to apply the algorithms to address multi-objective problems, we use an average weight aggregation due to its usefulness to the software engineering decision maker. The two complexity objective functions may produce different results for different anti-

patterns or different systems. Thus, the decision maker might select different weights for each objective. The proposed fitness function will combine CM and CM2 into one equation. This equation will assign a particular weight (w) to the first metric and (1-w) weight to the second metric. Since our objective is to minimize the complexity value, it is a minimization fitness function as can be seen in equation 3 below:

$$FF = \text{Min } (w^* K_1 *(\sum_{i=1}^{n} Use\ case_i)^2 + \sum_{i=1}^{m}\sum_{j=1}^{n} e_{ij} + K_2*[\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} - \sum_{i=1}^{m}\sum_{j=1}^{n} e_{ij}]$$

$$+ (1-w)\,[1-\sum u\ /\sum i + \sum a\ ] \qquad (3)$$

5.4 Preprocessing

Before running the search-based algorithms on the case studies, we have taken several steps to ensure the algorithm runs correctly. First, we converted the use case diagram to a suitable format and selected a two-dimensional array for the first case study and class objects for the second case study. Second, we determined the anti-pattern for which the algorithm is searching. Third, we determined the correction or the refactoring operation that the algorithm should perform in case it detects an anti-pattern. Finally, we specified the controlling parameters of the algorithm; for example, in SA, the controlling parameters are the cooling factor and the acceptance probability.

5.5 Algorithm Parameters

HC is a very simple greedy algorithm that has no parameters. LAHC is an improved version of LAHC that has a memory list. The size of the memory list has little effect on the algorithm [147]. For SA, the cooling factor is significant to the success of the algorithm in finding potential anti-patterns and refactoring it. If we set the cooling factor

too high, this means the algorithm will cool faster, which means it will terminate quickly, missing the opportunity to explore more of the search regions to find other potential anti-patterns. On the contrary, if we set the cooling factor too low, it takes a significant time to terminate even if all anti-patterns have been explored. Thus, many methods have been proposed in the literature to set the value of this parameter. For simplicity, we tried a basic ratio decreasing approach, where the temperature is reduced by a constant in each iteration.

5.6 Detection phase

In the detection phase, the algorithm selects a random use case to start with. Then, the algorithm checks if this use case has a design smell (anti-patterns) or not. There are many anti-patterns defined in the literature. El-Attar provided a comprehensive list of these anti-patterns [88]. Each anti-pattern has defined rules and conditions. The algorithm searches for these conditions, and if a match is found, this means the anti-pattern is detected. In this paper, for illustration purposes of the capability of the search-based algorithm in retrieving anti-patterns automatically, we focus on only one anti-pattern.

5.7 Refactoring phase

After the algorithm detects an anti-pattern in the detection phase, the algorithm corrects the design defect using a refactoring operation. After correcting the design defect, the algorithm calculates the complexity metric using an equation. If the complexity value is reduced, the algorithm continues moving to another use case. If not, then the algorithm moves to another use case if the threshold condition is not met.

5.8 Experiment Setup

The experiment was carried out using an Intel-Based computer powered by a 3.30 GHz processor and 4 GB memory. The code was implemented on a Windows 7 System using Java language.

The objective of these experiments is three fold: to provide an automated refactoring of use cases using HC, and SA; to compare the results between HC, LAHC and SA on two case studies using one quality metric and a combined use case quality metrics; and to compare between these three algorithms on a large use case. Three questions are proposed in this chapter:

RQ1: How can search-based algorithms automate the refactoring of use case models using a single quality metric? What is the performance of each algorithm? How can they be compared?

RQ2: How can search-based algorithms automate the refactoring of use case models using a combined quality metric? What is the performance of each algorithm? How can they be compared?

RQ3: Are these algorithms scalable to automate a large use case model using a combined quality metric? What is the performance of each one in a large use case model? Is the difference in performance between these algorithms significant?

To answer RQ1; we used two search-based algorithms: Hill Climbing (HC) and Simulated Annealing (SA) guided by one complexity metric on one use case diagram to

refactor a single anti-pattern. We ran three experiments to compare the performance of these two algorithms: in the first experiment, we used a uniform distribution, in the second experiment we used a normal distribution and finally, in the third experiment we used three algorithms to compare between the significance of the SA algorithm parameters in producing results. We use T-test and ANOVA statistical tests in discussing the results.

To answer RQ2; we used the same three algorithms: Hill Climbing (HC), Late Hill Climbing (LAHC) and Simulated Annealing (SA) and ran three experiments. These algorithms are developed and applied to two case studies on a use case diagram using two quality metrics. We ran the first experiment using the first metric alone, we ran the second experiment using the second metric only and finally we ran the third experiment using the combined metric.

To answer RQ3, we ran the three aforementioned algorithms on a large use case model containing 100 use cases and compared the results using Wilcoxon pair test.

In the first three experiments, the objective is to show the applicability of the approach; therefore, we used HC and SA only. In experiments 4 to 7, the objective is to compare the algorithms using a combined metric. Since HC is a greedy simple search-based algorithm, we added LAHC, which is a sophisticated variation of HC algorithm. The results show LAHC competes with SA in some experiments.

HC is a greedy algorithm and no parameter is set for this algorithm. LAHC has an expandable memory size and it is initialized to 0. SA initial temperature is $1 \text{ X } 10^9$. And in a case of a negative result, the temperature is reduced by 100.

5.9 Experiments and Discussion

In this section, we present the details of the experiments and the discussion in a response to each research question. We are going to apply a "Drop Function Decomposition Having Inclusion" refactoring operation as suggested by [25]. Each algorithm starts by picking a random number that represents a use case. If this use case exhibits an inclusion relationship, then the algorithm checks how many included use cases it has. If it has more than 2 included use cases, then an anti-pattern is detected and the algorithm is going to apply the specified refactoring operation. Afterwards, it calculates the quality metric ensuring that the correction of anti-pattern results in a better value of the metric, then the algorithm continues until a stopping criterion is met. If the anti-pattern is not detected or if the correction of the anti-pattern does not result in a better metric value, then each algorithm behaves differently. HC usually terminates quickly, LAHC continues depending on the memory list and the SA continues depending on the temperature value.

5.9.1 Experiments for RQ 1

In this section, we discuss the three experiments conducted to answer the first research question "How can search-based algorithms automate the refactoring of use case models using a single quality metric? What is the performance of each algorithm? How can they be compared?"

Experiment 1: Refactoring Use case Model using HC and SA (Uniform Distribution)

The initial complexity, without performing refactoring, of the used case study is 634 as calculated by CM1 metric. If HC or SA detects the anti-pattern and refactors it, then the

complexity value should be lower. The optimal value of refactoring all instances of the selected anti-pattern in this case study is 109 as calculated by the same metric.

The analysis of the obtained results shows an obvious superiority in favor of Simulated Annealing over Hill Climbing algorithms. Hill Climbing algorithm is known to be a naive greedy basic algorithm that gives good results quickly but it suffers largely of local optima. As it can be observed from figure 12, in some of the running instances of HC algorithm, the algorithm was not able to find any optimization path. On the other hand, Simulated Annealing in general is always able to find a refactoring sequence that optimizes the use case metric, as shown in Figure 13. This is an interesting observation over HC algorithm. Simulated Annealing however was not able, in any of the following instances, to reach the global optima.

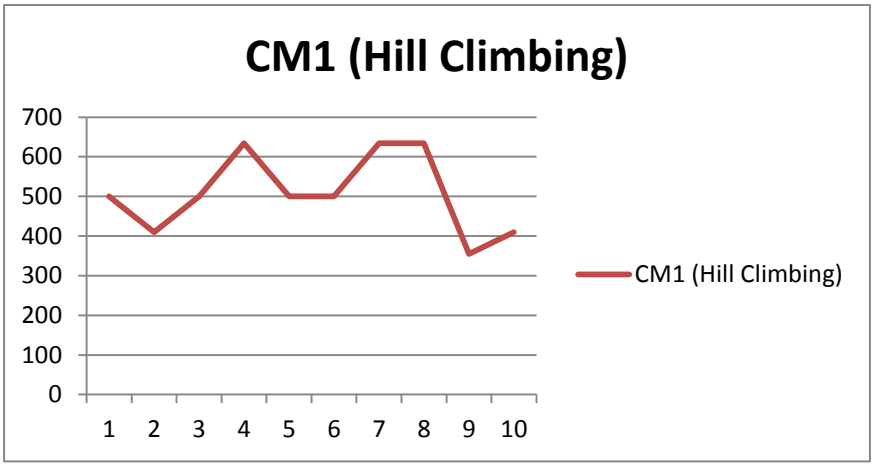

Figure 12: CM1 value using HC

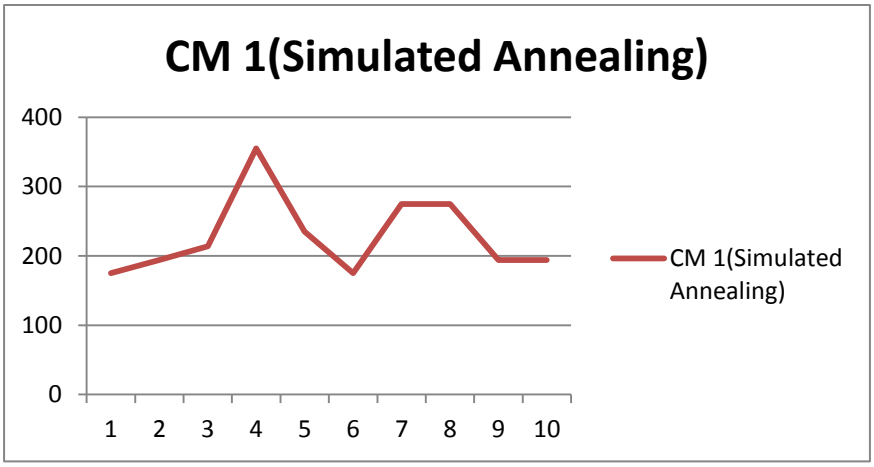

Figure 13: CM1 value using SA

Table 7 shows the values of CM1 metric using ten random numbers processed by HC and another ten random numbers process by SA algorithm.

Table 7: CM1 values of HC and SA

Run no.	CM1 (Hill Climbing)	CM1 (Simulated Annealing)
1	500	175
2	410	194
3	500	214
4	634	355
5	500	235
6	500	175
7	634	275
8	634	275
9	355	194
10	410	194

Figure 14 shows the normalized quality gain mean and standard deviation of the results returned by HC and SA. The figure shows that SA on average is able to obtain a quality gain of more than 60% with a very low standard deviation (less than 10%). On the other hand, HC cannot break the 20% barrier on average with a standard deviation that is closer to the mean (~ 20%).

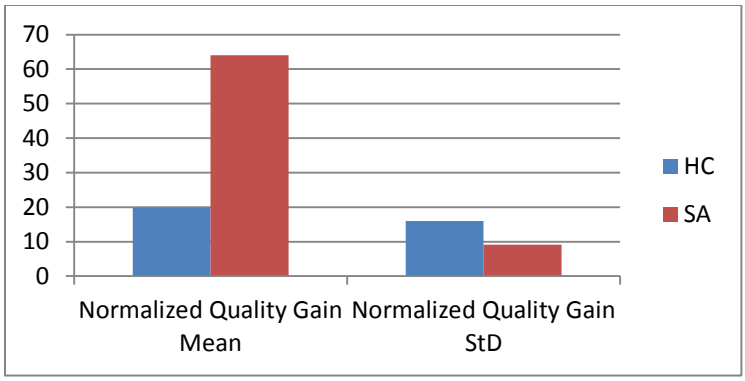

Figure 14: Normalized Quality Gain Mean and Standard Deviation

Experiment 2: Refactoring Use case Model using HC and SA (Normal Distribution)

Using Gaussian function in Java, we are able to generate a random number sequence that is normally distributed. The randomly generated sequence {3,1,3,0,7,3,2,4,4,3} is used for testing both Hill Climbing and Simulated Annealing.

Figure 15 shows CM1 value for both algorithms based on the generated normally distributed random sequence. The x-axis of the figure shows the starting position of the algorithm as generated by the Gaussian random function. The starting position refers to the use case number in the case study. Since we implemented the use cases using arrays and arrays in Java usually start from "0"; therefore, the use case counting starts from 0. Therefore, position 0 refers to the 1st use case. The y-axis shows the CM1 value without normalization, as explained earlier, the worst value of this metric in this case study is 634 if no refactoring was performed and the optimal value is 109 when all instances of the anti-pattern are refactored.

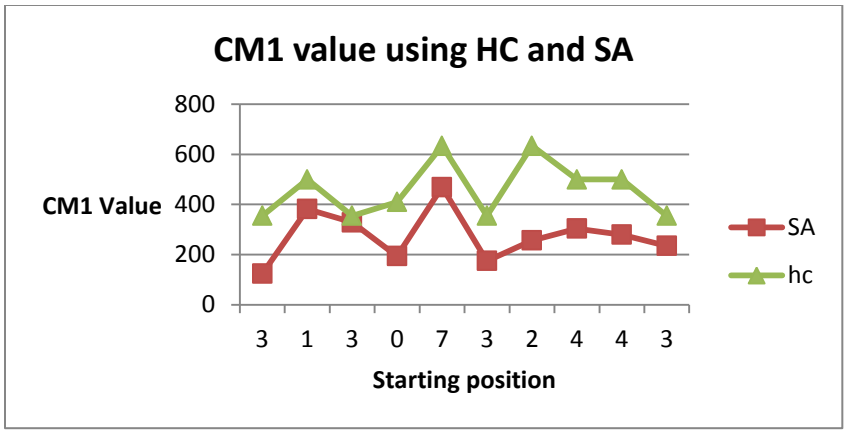

Figure 15: CM1 value using SA and HC over a Gaussian Generated Number

A number of interesting observations can be inferred from figure 15 . SA behaves better than HC across all running of this experiment. HC was not able to detect the anti-pattern when it starts from the point "7" (the 8th use case) and "2" (the 3rd use case) since CM1 value as indicated in the chart is still 634. HC terminates when there is no improvement and moves to another point if there is an improvement. Therefore, in this case, the 3rd use case and the 8th use case do not include the specified anti-pattern and thus the algorithm terminates quickly. This is a major limitation of HC algorithm. On the other hand, SA was always able to detect an anti-pattern and refactor it even when it reaches a use case that does not pose an anti-pattern such as the 3rd use case it is able to move to other ill-structured use cases and refactor them. Both HC and SA scored the worst when they start at point "7".

In comparison to uniform distribution, HC and SA both perform better when the random numbers are following a normal distribution. Table 8 shows the results of CM1 metric as computed by HC and SA in the two cases: when numbers are uniformly generated and when the random numbers follow a normal distribution. The mean and Standard deviation are also calculated.

Table 8: Comparison between Uniform and Normal Distribution of HC and SA results

Run No.	HC		SA	
	Uniform	Normal	Uniform	Normal
1	500	355	124	257
2	410	500	382	157
3	500	355	329	109
4	634	410	194	175
5	500	634	469	157
6	500	355	175	109
7	634	634	257	257
8	634	500	304	124

9	355	500	280	175
10	410	355	235	109
Mean	508	460	275	163
StD	100	112	102	56

We can infer from table 8 that normal distribution helps the algorithm to perform better. HC algorithm obtained a less average of CM1 using normal distribution rather than uniform distribution. In SA algorithm, the mean and standard deviation are improved significantly using normal distribution rather than uniform distribution.

We calculated T-test to ensure that there is a significant difference between the means of the two algorithms, as shown in table 9. Since T-test should be performed on a sample that is normally distributed; we applied the test on data in the normal columns for both HC and SA. At 95% confidence level, the p-value is 0.001138 < 0.05 and thus; it shows that there is a significant difference between the means of HC and SA.

Table 9: t-test of HC and SA

T-test: Two-Sample Assuming Unequal Variances		
	SA	*HC*
Mean	274.9	459.8
Variance	10461.43	12439.07
Observations	10	10
Hypothesized Mean Difference	0	
df	18	
t Stat	-3.8638	
P(T<=t) one-tail	0.000569	
t Critical one-tail	1.734064	
P(T<=t) two-tail	0.001138	
t Critical two-tail		2.100922

Experiment 3: Refactoring Use case model using SA with various cooling schedule

The discussion in the previous two sections shows the superiority of SA over HC. SA can be optimized by tuning two parameters: acceptance probability and the cooling factor. In order to study which parameter has the significant effect on the algorithm to perform better, we ran the algorithm using normal distribution generated randomly using various

cooling factors and acceptance probability. The results are shown in figure 16 .The results show that a higher cooling factor (cf =0.9975) usually results in a better value (lower value of CM1). This observation can be generalized over all acceptance probability(p). This observation is in alignment with SA theory that slows annealing (higher cooling factor) leads to better results.

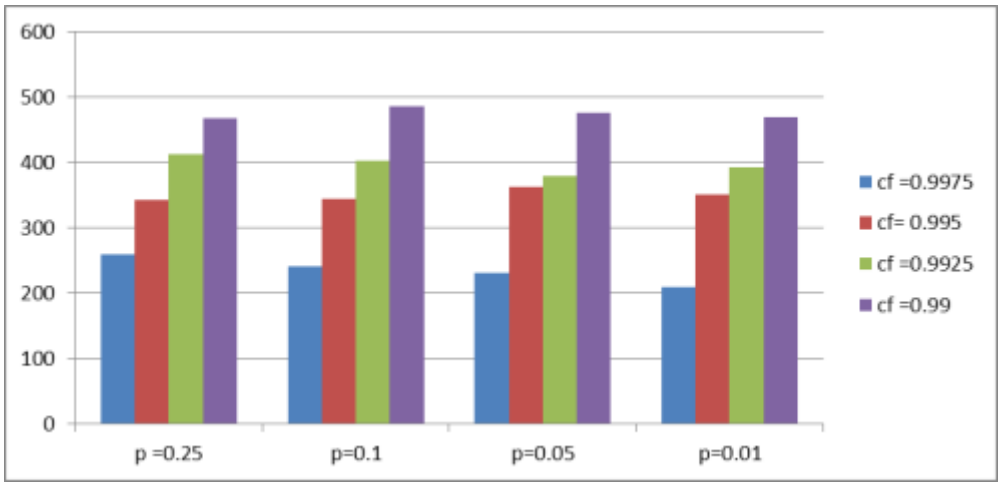

Figure 16: CM1 values using various values of (p) and (cf)

5.9.2 Expeiments for RQ2

In this section, we discuss the three experiments conducted to answer the second research question “How can search-based algorithms automate the refactoring of use case models using a combined quality metric? What is the performance of each algorithm? How can they be compared?"

Experiment 4: Refactoring Use case Model using HC, LAHC and SA (case study 1)

We ran each of the three algorithms: HC, LAHC and SA independently ten times on random numbers produced by a uniform distribution. The uniform distribution is the typical distribution generated by the random functions in Java and is the assumed behavior of the algorithm.

The initial complexity, without performing refactoring, for the first case study is 634 as calculated by the CM1 metric. If HC, LAHC or SA detect an anti-pattern and refactor it, the complexity value should be lower. The x-axis in Figure 17 to Figure 22 represents the mean of the experiments’ results. The y-axis represents the complexity metric value returned by the algorithm. In figure 17, the y-axis represents the CM1 value and in figure 18, it represents the CM2 value.

We can see from figure 17 that the SA algorithm has the lowest average of complexity values returned by the 10 random runs. HC’s average is the highest and LAHC is in between SA and HC. This observation can be attributed to the fact that SA tolerates non-good solutions. When a refactored use case by SA does not yield a better solution (lower complexity value), SA continues searching and running. This gives SA the flexibility to detect an anti-pattern and refactor it even if it starts from a use case that does not exhibit

anti-pattern symptoms. HC, on the other hand, is a greedy algorithm, so the algorithm terminates if a neighbor's solution is not better than the current solution. LAHC performed similarly but was better than HC on average. LAHC, maintains a fixed memory list of previous solutions. Initially, the memory list of LAHC is empty. The LAHC memory list is filled with good solutions, not with points. LAHC algorithm performance is improved when it is able to find many good solutions during the search to fill its memory list. However, if LAHC is unable to find any anti-patterns (good solutions), its memory list will be empty and hence, it will hinder the algorithm's ability to conduct useful comparisons. Thus, the LAHC will behave similarly to HC and terminates quickly; hence, its performance is better than HC.

When using CM2, as shown in figure 18 , SA still performs very well on average. LAHC was similar to HC but better than HC on average for the same reason explained above.

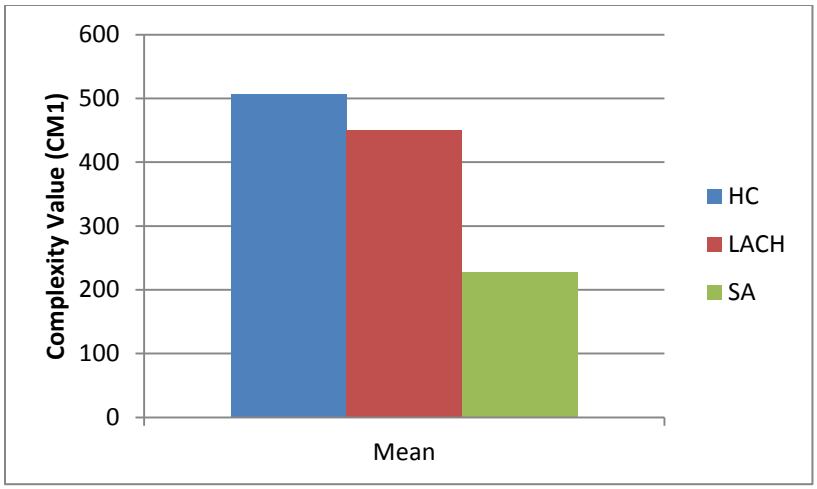

Figure 17: The average of CM1 value of case study 1

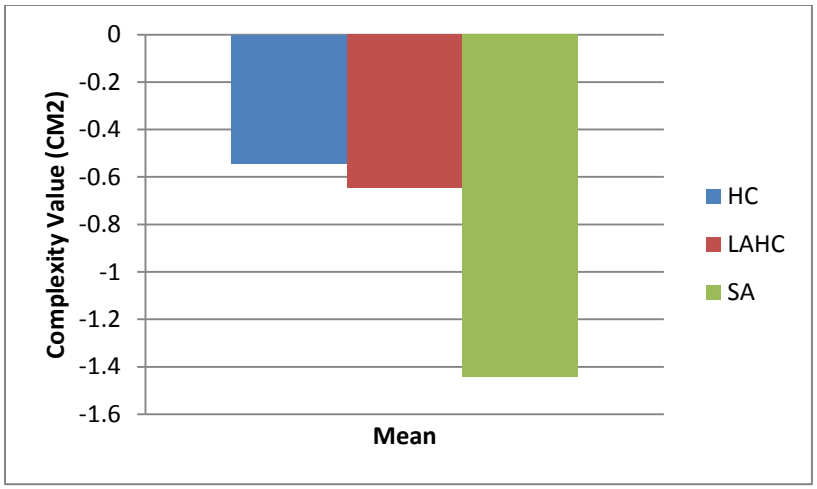

Figure 18: The average of CM 2 in case study 1

Experiment 5: Refactoring a Use case Model using HC and SA (case study 2)

This experiment was run using the three algorithms on a second case study to evaluate the two use case metrics CM1 and CM2 separately. The x-axis in figure 19 and figure 20 represents the average of the experiment results. Each algorithm was run 10 times. The Y-axis in figure 19 represents the CM1 value and in figure 20 ,it represents the CM2 value.

Similar to case study 1, SA performed well on average, while HC again performed adequately. LAHC's results on average are closer to SA than HC. LAHC is a more sophisticated hill climbing algorithm and it is expected to perform better than HC, which can be seen in the two metrics applied to case study 1.

Using the CM2 metric in case study 2, as shown in figure 15, both SA and LAHC performed very well, however HC's performance was incompetent. SA performed well in this case study using CM2. Of course, if we need SA to be optimal in all case studies, we can empirically tune the SA parameters until an optimal combination of the cooling factor and acceptance probability parameters is reached. However, this is not our intention in this chapter and is left for other research. LAHC performance is consistent with previous analysis. We can see that LAHC performance is better than HC and competes with SA. The variation of the average values of the three algorithms is smaller than in case study 1. This might be because case study 1 has more anti-patterns than case study 2.

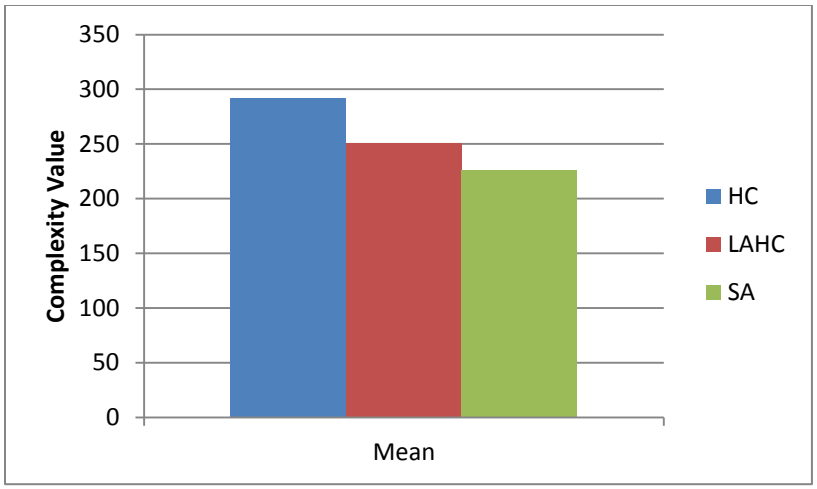

Figure 19: The average of CM1 in case study 2

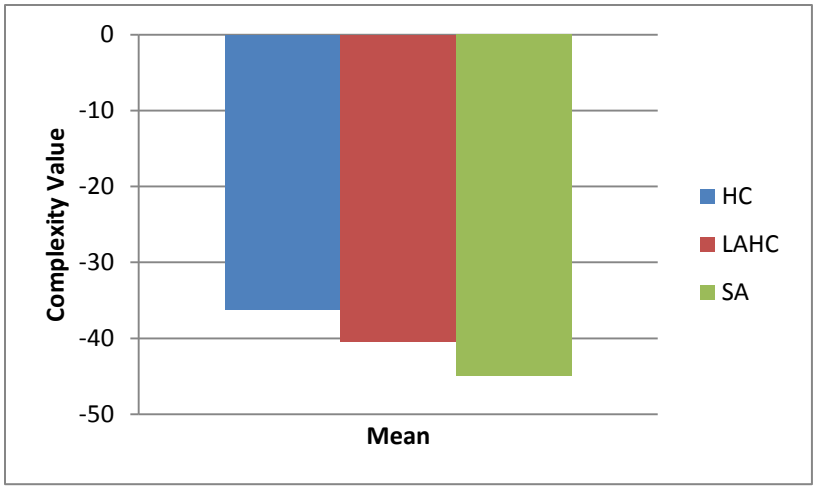

Figure 20: The average of CM2 using case study 2

Experiment 6: Refactoring Use case Model using HC, HACL and SA (combined-Metric)

The objective of this experiment is to evaluate the three algorithms using a multi-objective approach. The three algorithms will try to balance between the values of CM1 and CM2 metrics simultaneously. We set an equal weight (0.5) for each metric. In this experiment, we evaluate the algorithms using the two case studies. The x-axis in figure 21 and figure 22 represents the average of the experiment result. We ran each algorithm 10 times independently. The y-axis in Figure 21 and Figure 22 represents the combined values of CM1 and CM2 as in equation (2) for case study 1 and 2, respectively.

In this experiment, we combined the two metrics into one fitness function using a weight aggregation method. The initial fitness value for this case study considering the two metrics without performing any refactoring is 316.8. SA succeeded in scoring the best average. LAHC, as in the previous experiments, is between HC and SA. HC's average is the highest and this is in accordance with its results in case study 1.

As shown in Figure 17, the results are similar to those of the previous experiments. HC has the poorest performance, LAHC competes with SA and SA has the highest average. The variation of the average values in case study 2 is less than case study 1 for both when using a single metric or a multi-metric. We attribute this to the number of anti-patterns in this case study.

From the previous experiments, we can draw some interesting conclusions. HC is a greedy simple metaheuristic algorithm that has low performance in general. However, it

is a parameter-less algorithm, which makes it easy to implement. It might succeed in some runs in reaching a good fitness value.

We can also infer from the experiment results that SA is a stable algorithm. It gives a good performance across all metrics, case studies and runs. It might perform worse than the greedy HC in some runs; however, it also can compete with the other algorithms in obtaining the optimal fitness value. Nevertheless, SA is a parameterized search-based algorithm. Two operators or parameters affect its performance: the cooling factor and the acceptance probability. There is no optimal setting for these two parameters and their values depend on empirical analysis, problem type and designer's experience.

The performance of LAHC is good in general and results in better values than HC on average and is sometimes closer to SA as in case study 2; however, unlike SA, LAHC is a parameter-less algorithm, which makes it easier and more suitable for novice designers.

From the above discussion, we can conclude that if the case study is small and the designer can afford the cost of many runs and the objective is to obtain a fast refactoring result, then HC can be a good choice. If the designer is interested in having a general simple algorithm that is able to perform quite well given the fact that several runs of the algorithms can be afforded, then LAHC is a good candidate. If the objective of the designer is to have a stable algorithm that can give a good result from only one run, then SA is a good choice. However, the parameter settings must be set optimally, either analytically or empirically.

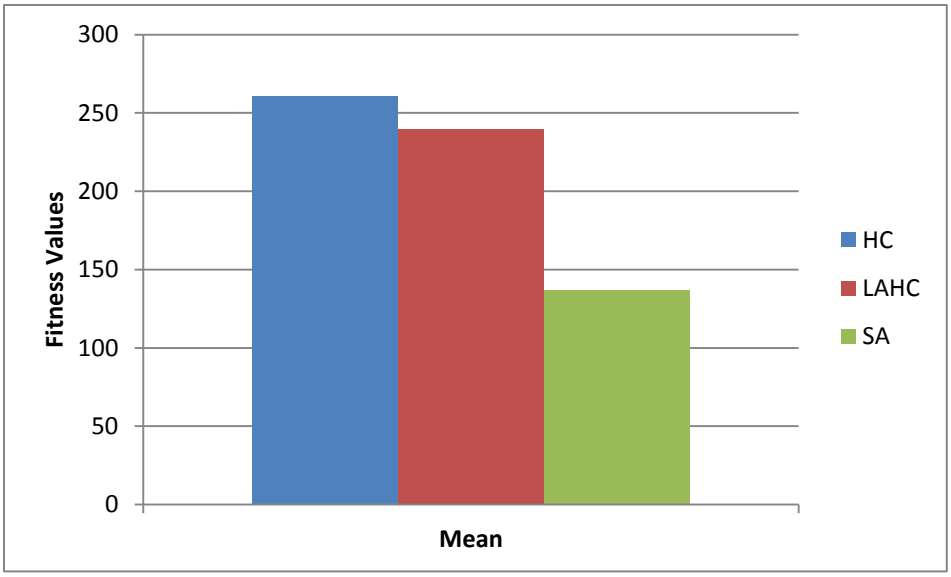

Figure 21:The average of Fitness function using case study 1

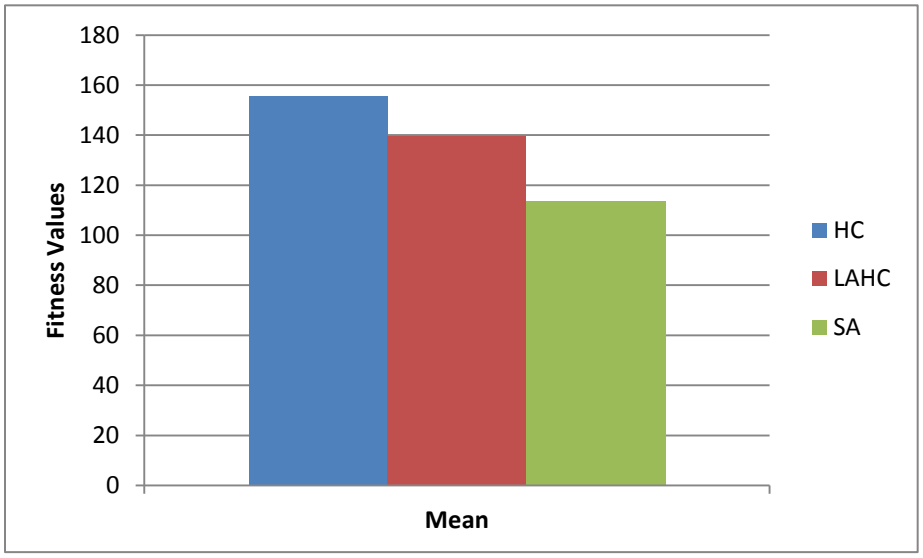

Figure 22: The average of fitness function using case study 2

5.9.3 Experiments for RQ 3

In this section, we discuss the experiment conducted to answer the third research question "Are these algorithms scalable to automate a large use case model using a combined quality metric? What is the performance of each one in a large use case model? Is the difference in performance between these algorithms significant?

Experiment 7: Refactoring a large Use case Model using HC, HACL and SA (combined-Metric)

We ran an experiment with a large dataset of 100 use cases. This is quadruple the second use case. The purpose of this experiment is to show the scalability of each algorithm to a large dataset. Then, we to assess these algorithms statistically using Wilcoxon tests as shown in table 11.

Table 10: Wilcoxon pair signed test

HC	LAHC	SA
2392.5	2107.48	2247.74
2343.75	2247.74	2392.50
2541.76	2392.50	2247.74
2392.50	2107.48	1797.69
2108.27	2392.50	2541.76
2392.50	2107.48	1840.45

2541.76	2247.74	2061.72
2392.50	2541.76	2107.48
2541.76	2107.48	1113.53
2108.27	2247.74	1927.46
2541.76	2247.74	2392.50
2392.50	2541.76	1927.46
2392.50	2343.75	1713.68
2541.76	2247.74	2541.76
2541.76	2343.75	2107.48

We ran Wilcoxon pair signed test on each pair of the above algorithms. The p-value is 0.026 between HC and LAHC; that means the results is significant at 0.05 levels. Between HC and SA, the p-value is 0.005. It is significant at 0.01 levels. In applying Wilcoxon between LAHC and SA, the p-value is 0.026 which is significant at 0.05 levels. Therefore, these algorithms are scalable on large use cases.

5.10 Threats to Validity

In this section, we present some threats that may affect the validity of this research and our mitigating actions to reduce or remove the impact of each threat.

One threat is that HC, LAHC and SA algorithms are intrinsically based on randomness; that is, at each iteration, the algorithm selects a different random initial point. Some points result in better optimization values than others. In this regard, we opt to run the algorithm several times and record the optimization value at each run. We ran the algorithms many times and applied statistical tests such as the mean or t-test.

Another threat is that our quality measurement is based on two published metrics that measure the complexity of use case diagrams. Other metrics might produce different results or imply different performance of these algorithms. Therefore, we applied our experiments on the two metrics separately, and then we ran another experiment combining these two metrics and recorded the results.

Our results are based on adapted published case studies of use case diagrams. Other case studies might produce different results, especially when different refactoring operations are applied. To mitigate the impact of this threat, we ran each algorithm 10 times and recorded the average.

Finally, the HC, LAHC and SA algorithms are implemented by the authors and rigorous testing was undertaken to ensure the correctness of the developed code.

5.11 Conclusion and Future Work

Use case refactoring is an important step in the software design process since defects in this stage might cause major problems to subsequent stages. Manual refactoring is costly and time consuming. Using search-based algorithms to search for any possibility of a poorly structured design to refactor it using a specific refactoring operation is useful. Most of the search-based algorithms were done on code refactoring. In this paper, we

used search-based algorithms to automatically perform refactoring on the use case diagram. The results show an interesting and promising output by implementing three search-based algorithms namely: HC, LAHC and SA.

SA performs very well in all experiments. From experiment 3, we infer that cooling factor contributes to the result the most. In experiments 5 to 7, we see SA has a stable performance using one or combined quality metric and to large use case as well. HC simplicity does not prevent it from obtaining good results quickly. HC requires no parameter tuning; however, the designer needs to run it many times to obtain an acceptable result. LAHC is a win-win algorithm. It is as simple as HC but it gives a comparable result to SA. It is able to get excellent results in some runs; however, on average, LAHC performance is closer to HC than SA as can be shown from figure 14, figure 15 and figure 21.

Our future work includes applying other search-based algorithms on various case studies using different refactoring operations. We also plan to investigate applying search-based algorithms to other UML models.

CHAPTER 6

SEQUENCE DIAGRAM REFACTORING

In this chapter, we use a hybridized algorithm to perform refactoring on sequence diagrams. Our hybridized algorithm is named KSA as an abbreviation of the combination of Kmeans and SA [148].

To illustrate how our KSA algorithm works, we explain the process of each algorithm separately and then we explain the hybridized process. Figure 23 shows the steps for the Kmeans algorithm to refactor a sequence diagram. As illustrated in the figure, there are two major preprocessing steps required for this algorithm, which are: extracting entities and features and constructing a similarity matrix.

Figure 24 shows the steps for the SA algorithm to refactor the sequence diagram. The sequence diagram should be converted into a medium representation. Since our objective is to hybridize SA with Kmeans, we used the same representation for both of them.

Figure 25 shows the steps of the proposed KSA algorithm. The details of this algorithm are explained in detail below.

In the first step, the sequence diagram is converted into an entity-feature matrix. After this, the data mining algorithm (Kmeans) will use the constructed entity-feature matrix to create clusters. Each cluster groups the most similar objects together. Therefore, in this paper, each cluster contains messages that have similar features.

In the next step, we have a collection of clusters where each cluster groups the most similar messages together. This is the initial solution on which the second algorithm will run. The search-based algorithm (SA) will start from within the clusters returned by the Kmeans algorithm instead of starting randomly from any position. SA will be guided by metrics to refactor the sequence diagram. The refactoring operation will be “moving messages”. This is similar to the “move method” in code refactoring. The algorithm will apply the sequence of this refactoring operation on random points of the cluster returned by the Kmeans algorithm.

Finally, the hybridized algorithm will provide suggestions on sequence diagram refactoring using the “move message” operation. The cohesion and the coupling metric will be calculated after the suggested refactoring operations in order to evaluate the algorithm.

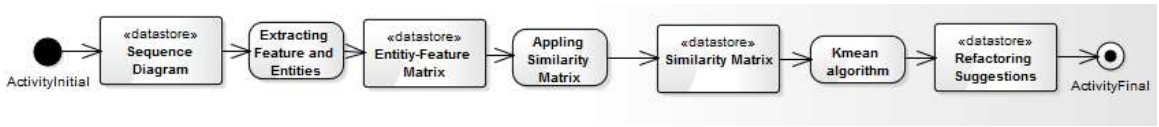

Figure 23: Kmeans for sequence Diagram

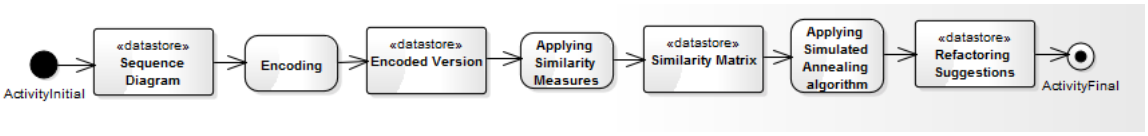

Figure 24: SA for sequence diagram

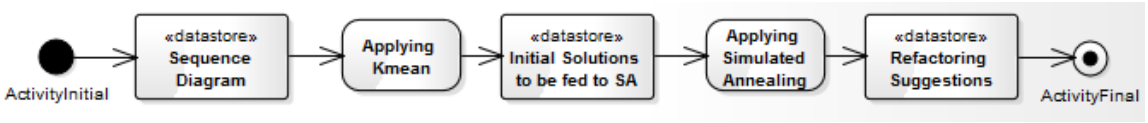

Figure 25: KSA for sequence diagram

There are several general limitations in Kmeans clustering algorithms: 1) the Kmeans algorithm tends to get stuck at a local optimum point; 2) the number of clusters must be specified in advance; 3) the random initialization of cluster means can be far from all points, and 4) the Euclidian distance is based on points in two-dimensional planes, and the entity-feature matrix must be constructed. These limitations are also applicable to problems in software engineering domain.

Likewise, SA has several limitations similar to other search-based algorithms: 1) the SA algorithm tends to get stuck at a local optimum point; 2) the first initial number might be far from the optimal solution; 3) it requires a fitness function; and 4) it has some controlling parameters such as temperature and probability that must be tuned.

The KSA hybrid algorithm seeks to optimize the performance of the SA algorithm by providing better initial points of SA using the Kmeans algorithm. The Kmeans algorithm will run first on the problem where it will extract the points that have a potential improving value and gather them into one cluster. This cluster then will be passed to the SA algorithm to start randomly from any position within this cluster. This will improve the performance of the SA algorithm by letting it start from a promising point in the search space. Then, at each iteration, the SA algorithm will move to another point within this passing cluster in order to ensure that the SA algorithm proceeds from one promising point to another promising point and reduce the chance of being trapped in local optima. In addition, this communication between the Kmeans algorithm and the SA will save many iteration where SA will investigate weak points that will slow the algorithm convergence.

6.1 Entities and Features

Clustering algorithms depend on grouping entities together, based on the similarity value found in their features. It is important to select a number of features that reflect the similarity between entities. Selecting too many features may result in clustering each entity in a separate group. Likewise, selecting too few features may end up with crowding a few clusters with many entities that do not relate to each other. Hence, the selection of features should be considered when designing clustering algorithms to solve a particular problem.

Entities are the objects that we want to cluster. In our proposed solution, a sequence diagram can be treated as a set of entities with different features. Our objective is to refactor sequence diagrams using a hybridized algorithm and evaluating the results using selected quality metrics. To achieve this, we are going to group similar messages in the most suitable class. For sequence diagram refactoring, methods are considered entities. Each message has two features 1) the name of the function it is sending to or receiving from; 2) message type (a direct or an iterative message). In the entity-feature matrix, we record how many features of the message each class shares, which can be none, one or more features. In our case study, we used the method parameters as the message features. However, our algorithm works on any other representations of features.

In a sequence diagram, cohesion can be defined as the number of messages a class is able to access within itself. The highest cohesion is required since it will reduce the number of communication messages with other classes. There are different metrics for measuring cohesion. We selected LCOM2 [149] as the cohesion metric as in [94] for comparison purposes and for its applicability to our proposed solution that is based on similarity.

LCOM2 is found to be more suitable for our study because it considers calculating the shared attributes between methods; other cohesion metrics are found to be less relevant. For example, LCOM3 and LCOM4 do not consider the number of attributes that are shared between two methods [150]. Loose Class Cohesion (LCC) represents the connection between public methods without considering the sharing of instance variables, so similarity between methods are not considered which rendered the usefulness of clustering algorithm.

We define coupling as the number of direct messages the class is sending to or receiving from other classes. Reducing the coupling value is desirable due to the fact that it will reduce the communication messages between classes.

A higher similarity of features indicates similarity of functionality, which in turn increases cohesion and reduces the coupling of message communication between different classes. The advantage of such processes is to increase the quality of the model, as cohesion and coupling are desirable features in object-oriented software systems.

6.2 Similarity Matrix

A similarity matrix is a matrix where rows represent features and columns represent entities. The value inside each matrix cell represents how many features there are in one entity. The application of the Entity-Feature matrix for software refactoring was originally proposed by Lung et al. [151] .

Since clustering algorithms are based on similarity distance, clustering algorithms are applied for software refactoring to enhance cohesion and coupling. Cohesion implies that if a class has many similar features of a particular method, then that method should be a member of that class. Thus, by knowing which class has the most similarity with a method, we can move that method to that class. Similarly, coupling refers to the communication between two different entities. By knowing which method should belong to which class, we can reduce the number of communication messages between these entities. Software developers aim to increase the cohesion of the software and minimize its coupling.

Table 12 is adopted with some modifications from Alkhalid et al. [94] to compare with the authors' results. In their paper, Alkhalid et al. applied only clustering algorithms to refactor software classes. In this chapter, we use a hybridized algorithm. We use this table for comparison purposes and due to the scarcity of sequence diagram data that is representative enough in order to show the benefits of the proposed approach.

Figure 26 shows the sequence diagram example represented by Table 12. As we can see, we have four classes and sixteen messages. Variables belongs to class1 are named by the letter 'a' followed by a sequence number. Likewise, variables of class2, class3 and class4 are named by the 'b', 'c' and 'd' respectively followed by a sequence number. Each message contains parameters. These parameters are the features that we are looking for in order to apply the algorithm. For instance, Message 1 denoted by M1 in the diagram representing a message that is sent from class2 to class1 and it has three parameters: a1, b1, and b2. The parameter variable a1 belongs to class1 and the parameter variables b1

and b2 belongs to class2. Message2 represents a method that calls itself and thus in this case it has only the variables named with ‘a’.

Table 12 shows the entities and features of a sequence diagram case study. Value 1 in the intersection of the Class 1 column and Message 1 row indicates that class 1 has only one feature of message 1 while class 2 has two features of the same message. Class 3 and class 4 do not share any features with this message. Thus, to increase cohesion, the algorithm will propose moving message 1 to class 2.

Table 11: Similarity Matrix between Entities and Features

	Class 1	Class 2	Class 3	Class 4	Array Index
Message1	1	2	0	0	0-3
Message2	2	0	0	0	4-7
Message3	2	0	0	0	8-11
Message4	2	0	0	0	12-15
Message5	0	2	0	0	16-19
Message6	0	0	1	2	20-23
Message7	1	2	0	0	24-27
Message8	0	0	0	2	28-31
Message9	0	0	2	0	32-35
Message10	0	0	2	0	36-39
Message11	0	0	2	0	40-43
Message12	0	0	2	0	44-47
Message13	2	1	0	0	48-51
Message14	1	0	2	0	52-55
Message15	0	0	0	2	56-59
Message16	0	0	0	2	60-63

Table 13 shows the messages in each class. This is similar to the adopted case study. However, in the adopted case study, the relationships are between classes and methods.

In this paper, we modify the relations to be between classes and messages. Therefore, we have 4 classes and 16 messages where each class has 4 messages. Table 13 and 14 show the initial cohesion and coupling values of each class respectively, while table 15 indicates the necessary refactoring operations in order to reach an optimal state in terms of cohesion and coupling metric values.

Table 12: Class and its belonging messages

Class	Sending Messages
1	1, 2 ,3 and 4
2	5, 6, 7 and 8
3	9, 10, 11 and 12
4	13, 14, 15 and 16

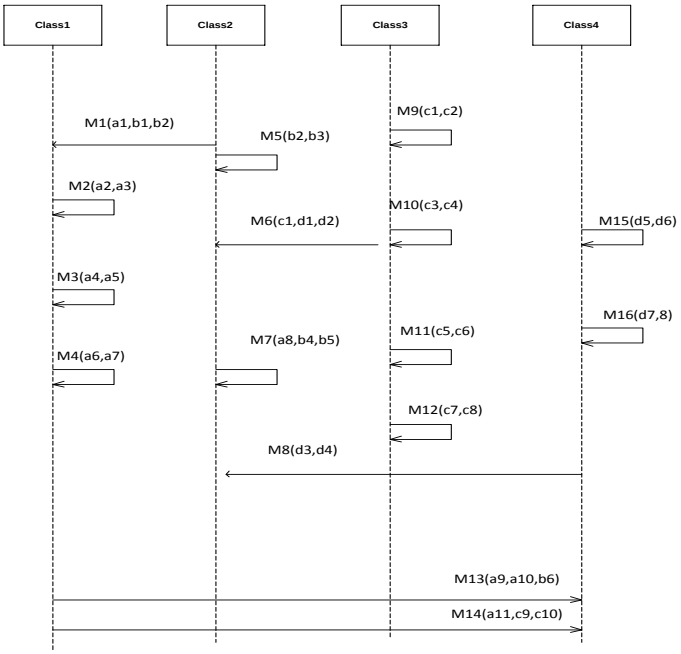

Figure 26: Sequence Diagram Case study

The cohesion values are generated using the LCOM metric. To calculate the LCOM of a class: first, count the number of messages that share features, denoted as (Ms). Then, calculate the number of messages that do not have any features in common, denoted as (Mn). If (Mn) - (Ms) > 0, the LCOM is (Mn –Ms). Otherwise, LCOM is zero. So LCOM can be zero or bigger. Table 14 shows the value of cohesion for each class based on the LCOM metric.

Table 13: The cohesion value of the classes in our case study

Class	Relations between messages	Cohesion Value
Class 1	M1 ∩ M2, M1 ∩ M3, M1 ∩ M4, M2 ∩ M3, M2 ∩ M4, M3 ∩ M4	0
Class 2	M5 ∩ M6, M5 ∩ M7, M5 ∩ M8, M6 ∩ M7, M6 ∩ M8, M7 ∩ M8	0
Class 3	M9 ∩ M10, M9 ∩ M11, M9 ∩ M12, M10 ∩ M11, M10 ∩ M12, M11 ∩ M12	0
Class 4	M13 ∩ M14, M13 ∩ M15, M13 ∩ M16, M14 ∩ M15, M14 ∩ M16, M15 ∩ M16	3

Table 15 shows the coupling values of the sequence diagram before refactoring. In this chapter, coupling refers to the coupling between classes. It indicates the number of messages communicated between two classes. For instance, Class 1 has a coupling value of 1 because it has only one message (Message 1) that is communicated to another class (which in this case is class 2).

Table 14: The coupling classes of our case study

Class	coupling
Class 1	1
Class 2	2
Class 3	0
Class 4	1

Table 16 shows the number of refactoring operations that are required to move the system into an optimal state in terms of cohesion and coupling. In this paper, we use the “move message” operation, which is similar to the “move method” operation used by Alkhalid [94]. As shown in table 16though classes 1, 2 and 3 already have an optimal LCOM value of zero, they are still involved in the refactoring process. Applying refactoring on class 4

only might reduce the LCOM of class 4, but increase the coupling or even the LCOM of the other classes. Thus, balancing both metrics is essential.

Table 15: The number of refactoring candidates

Class	Required Refactoring operation
Class 1	1
Class 2	1
Class 3	1
Class 4	2

6.3 Results & Discussion:

In this section, we show the results of our two experiments: one using the SA algorithm and the second using our hybridized KSA algorithm. The experiment was carried out using an Intel-based computer powered by a 3.30 GHz processor and 4 GB memory. The code was implemented on a Windows 7 system using Java. The parameter setting of both algorithms is:

Kmeans has two parameters: the number of clusters and the randomized function for cluster's center. Since we have four classes in our case study, we set the number of clusters to be four; the center of each cluster is determined randomly using the following function:

$C_k = 1 + 9.00 *$ rand.nextDouble(); where C represents the center and k ε {1,4}

SA has two parameters to be set: initial temperature and the Cooling Factor. The cooling factor is fixed in all experiments, but the initial temperature is changed. The parameter settings of SA is shown in the result sections. These parameters are set arbitrarily using trial and error until this setting found to return good results.

The objective of this experiment is to answer RQ1 “Is there an improvement gained by hybridizing two algorithms: one from the “search-based” category and the other from the “data mining” category as compared to implementing a single algorithm?”

6.3.1 Experiment 1: Refactoring Sequence Diagram Model using SA

We implemented an SA algorithm on the above case study to provide suggestions to the users on how to move messages between the different classes in order to maximize cohesion. This is a demonstration experiment to show the limitation of the SA algorithm. However, when we run our hybridized algorithm, the four selected metrics are discussed. The SA algorithm is based on a random initialization of one point. The algorithm can randomly pick any point out of the 64 points shown in table 12. After this, the algorithm determines the class to which the selected message should belong. Then, based on similarity features, the algorithm will recommend either moving the message to another suitable class or recommend to keep the selected method in the same class. After all the recommendations of the SA algorithm, the LCOM value is calculated. LCOM is a cohesion metric, hence reducing the value of LCOM is desirable. Therefore, in the above example, Class 4 has an LCOM value of 3 which needs to be reduced.

Running the SA algorithm returns this sequence of random numbers (57, 55, 37, 38, 25, 2). The first number is 57, which corresponds to the array index 57 in Table 12. Here, 57 corresponds particularly to the number of features that “Message15” shares with “Class2”. As shown in the table, there is no common feature between “Class2” and “Message15” and subsequently, picking up this point will make no contribution to reducing cohesion or increasing coupling. In this case, the SA algorithm will pick another point randomly if the probability threshold is not met yet.

The next point the algorithm selects is 55. Again, 55 corresponds to the number of features in common between “Class4” and “Message14” which in this case is null. Therefore, picking up this point will not improve cohesion or coupling and the algorithm continues to pick another point as long as the probability threshold is still satisfied. The third point is 37, which also does not have any impact on the cohesion or coupling value. The same applies to all remaining points.

In another run, the following sequence of numbers was generated (3, 61, 40, 23, 29, 2). In his sequence, point 23 corresponds to the number of features in common between “Message6” and “Class4”. According to Table 12, Message6 initially belongs to “Class2”, though it is more similar to “Class4”. Therefore, the SA will recommend moving “Message6” to “Class4”.

Limitation of this approach

The algorithm will pick any point randomly. The algorithm might start from a very bad point such as the “0” point in Table 12. This will affect the algorithm’s performance since the algorithm might waste some iterations picking up the “0” value which is not useful because it indicates no common feature of a message in a class. In addition, the existence of these bad points might converge the algorithm quickly to local optima. Consider the situation where the algorithm picks this sequence of values randomly (2, 0, 0, 0, 0, 0, 0, 0, 0, 0). In this sequence, the algorithm starts from a good point where it has a maximum similarity of features, but it keeps going to one weak neighbor to another. After a few iterations, the acceptance probability will be low and the algorithm will hit the condition criteria, forcing it to terminate.

6.3.2 Experiment 2: Refactoring Sequence Diagram Model using a hybridized SA and clustering algorithm

We propose the hybridized SA algorithm and the Kmeans using a pipeline fashion [152]. Pipeline hybridization means that algorithm "A" runs fully and its results are taken to algorithm B as inputs. This type of hybridization has been investigated in several papers concerning Kmeans and SA algorithms [153-155]. The clustering algorithm (Kmeans) will take all the points (messages) and cluster them based on their high similarities with the corresponding classes. Then, the SA algorithm will save time by picking points that have the potential to reflect on the sequence diagram refactoring.

When we run our hybridized KSA algorithm, the following sequence of points appear (38, 49, 4, 25, 1, 1, 59). This is the final results of points picked up by our hybridized algorithm. Before we delve more deeply into analyzing these results, let us look at the intermediate results. The KSA algorithm starts by providing four cluster centers randomly since we have four classes. These centers are at the points {4.3, 6.3, 9.3, 3.9}. Then the algorithm continues updating the centers of the clusters until all points are assigned to one cluster which includes all points that might make a contribution to the cohesion metric, that is, the points that indicate that there are similar features between messages and classes. Now, the KSA will pick up points out of this cluster to ensure that any point taken should have a similarity value and hence it might, but not necessarily, reduce cohesion or increase coupling.

If we check all the points in this sequence as returned by KSA: (38, 49, 4, 25, 1, 1, 59), we find that all these points have similarity values. Unlike running SA alone where SA might go for many iterations picking up non relevant points (points with no similar features values), KSA only picks the right point, thus increasing the speed of the algorithm.

Table 16: Results of first run of KSA

No.	Point	Representation	KSA recommendation
1	38	Message10, Class3	Keep it
2	49	Message13, Class2	Move Message13 to Class1
3	4	Message2, Class1	Keep it
4	25	Message7,Class2	Keep it.
5	1	Message1,Class1	Keep it
6	1	Message1,Class1	Keep it
7	59	Message15,Class4	Keep it

Metric /Class	1	2	3	4		Recall	20%
LCOM	0	0	0	2		Precision	20%
Coupling	1	1	1	1		Ratio	7/ 1=7
Initial Temperature of SA	10000000.0		Termination Condition	Temperature > 1		Cooling Factor	Temperature /10.0

The following details show the answer of RQ2 “How to evaluate the effectiveness of the hybrid algorithm in refactoring sequence diagrams using quality metrics such as cohesion (LCOM2), coupling and recall and precision measures?”

Table 17 shows the values of LCOM, coupling, recall, precision and the ratio. Moving message 13 to class 1 will reduce the cohesion of class 4 to 2 where its initial value is 3, as shown in Table 14. In addition, the coupling of class 4 is reduced to 1 where its initial value is 2, as indicated in Table 15; however the value of LCOM for class 1 is still 0. This is an optimal value of cohesion and moving the message leaves it in this state. Meanwhile, the KSA algorithm recommends one refactoring operation out of 5 required

operations to lead the system into an optimal state. So the recall is 1/5 or 20%. The recommendations provided by the algorithm are correct, so again the precision is 1 correct refactoring operation out of 5 which is 20%. The ratio indicates how many iterations result in the refactoring operation in comparison to the number of iterations the algorithm undertakes. In this run, the algorithm runs for seven (7) iterations where only one iteration recommends a refactoring operation. So the ratio is 7. The ratio can tell us how good the algorithm is for finding the good results. A ratio of 7 is not considered a good value since the algorithm has to waste six other iterations to find one good point in one iteration.

All recommendations of KSA are correct and whether it recommends moving messages or leaving them in their original class, the algorithm is able to determine the class for each message. Our case study involves four classes with three classes have a good cohesion value and the fourth having a high cohesion value. This is a difficult scenario for the algorithm since picking 75% of space points will not result in a good value. Thus, we run another experiment by relaxing the probability threshold to allow the algorithm to run for a few more iterations.

In the second run, as shown in Table 18, we relaxed the acceptance probability so the algorithm can continue for more iterations than the first run before it terminates. In this run, the algorithm picked 11 random points recommending 4 refactoring operations. So the recall of the algorithm in this run is 4 out of 5 or 80%. All the recommended refactoring operations by the algorithm are correct, so the precision is 80% too. The ratio is considerably good; the algorithm runs 11 iterations to find 4 refactoring operations, which means the ratio is 2.75. This is far better than the first run. We did not calculate

coupling and cohesion in this run until all necessary refactoring operations are recommended by the algorithm.

Table 17: Resutls of second run of KSA

No.	Point	KSA recommendation
1	4	Keep it
2	12	Keep it
3	46	Keep it
4	0	Move Message1 to Class2
5	31	Move Message8 to Class4
6	8	Keep it
7	22	Move Message6 to Class4
8	23	Keep it
9	17	Keep it
10	49	Move Message13 to Class1
11	22	Keep it
Recall		80%
Precision		80%
Ratio		11/ 4=2.75
Initial Temperature		100000000000.0

In the third experiment, as shown in Table 19, the KSA algorithm was able to recommend all necessary operations in order to reach the optimal state, but it has to run for 21 iterations. The LCOM and coupling values are now optimized. The LCOM value of class4 now is 0 and the coupling of the four classes are 0 too. When the LCOM value is decreased, it indicates better cohesion. The value of the coupling metric decreases as well which is a desirable result too. The recall value in this experiment is 5 out of 5 which is 100%. All of these recommendations are correct, so the precision is 100%. However, the ratio is 4.2 since the algorithm takes 21 iterations to find 5 correct operations. This means that the algorithm, on average, has to go for 4 iterations to find one good refactoring. Figure 27 shows the refactored system as recommended by our algorithm.

Table 18: Results of third run in KSA

No.	Point	KSA recommendation
1	4	Keep it
2	12	Keep it
3	46	Keep it
4	0	Move Message1 to Class2
5	31	Move Message8 to Class4
6	8	Keep it
7	22	Move Message6 to Class4
8	23	Keep it

9	17					Keep it	
10	49					Move Message13 to Class1	
11	22					Keep it	
12	34					Keep it	
13	49					Keep it	
14	38					Keep it	
15	54					Move Message14 to Class3	
16	38					Keep it	
17	12					Keep it	
18	31					Keep it	
19	23					Keep it	
20	48					Keep it	
21	1					Keep it	
Metric /Class	1	2	3	4		Recall	100 %
LCOM	0	0	0	0		Precision	100%
Coupling	0	0	0	0		Ratio	21 / 5 = 4.2
Initial Temperature	100000000000.0						

In comparison to [94], where they applied only clustering and class4's LCOM was decreased by 2, in our experiment, class4 was decreased by 3. Moreover, they used the Coupling Through Abstract Data Type (CTA) as a coupling metric. The CTA value for class1 is increased by 1, for class2 it is decreased by 1, there is no change for class3 and a decrease by 3 for class4. In this chapter, we opt to select a coupling between the messages' metric due to the difficulty of applying CTA in sequence diagrams. The coupling metric in our work was decreased by 1 for all four classes.

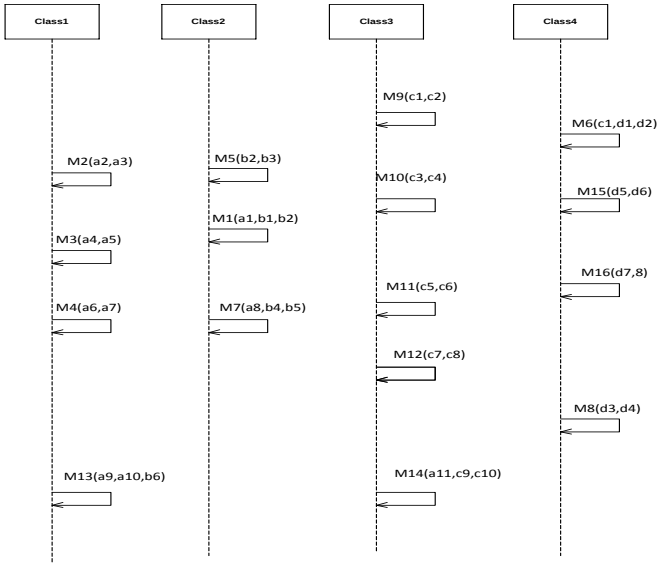

Figure 27: The refactored Sequence Diagram

6.3.3 Comparison with KHC

Table 20 shows the results returned by running a hybridized algorithm of k-mean and Hill Climbing (HC) algorithms. We run the experiment eight times and recorded the results. The table shows the experiment number, the picked random point and the algorithm's recommendation. In the first five experiments, the algorithm does not find any refactoring opportunity and that is why it terminates after one run. In the sixth experiment, the algorithm finds two refactoring opportunities. The first refactoring opportunity reduces the cohesion and thus allows the algorithm to go for iteration. Though the second iteration results in recommending a refactoring opportunity but since it does not reduce the cohesion metric, the algorithm terminates. In the eighth's run, we see a similar phenomenon. Though the algorithm is able to find a refactoring opportunity, but since this refactoring has no impact on the cohesion metric, the algorithm terminates.

In comparison with SA, HC is very inefficient in working in such difficult case study. SA's strength in relaxation on the termination condition allowing for more iterations even if the refactoring opportunity is not found or the cohesion metric is not reduced, makes it more suitable than HC algorithm. Nevertheless, hybridizing HC with k-mean has its merits over running HC alone. As explained in the KSA section, hybridization directs HC to pick useful points (points with nonzero values as shown in Table 20.

Table 19: Results of KHC

Experiment No.	Point	KHC Recommendation
1	46	Keep it
2	38	Keep it

3	8	Keep it
4	38	Keep it
5	12	Keep it
6	23	Move Message 6 to Class4 Move Message13 to Class1
7	51	Keep it
8	54	Move Message14 to Class3

6.4 KSA algorithm for Extract Message Refactoring

In this experiment, we show how KSA algorithm is applicable in refactoring extract message operation. Extract message operation is similar to the extract method in code refactoring. If the message has features that belong to more than one class, then we will create a set of new messages where each message contains features belonging to one class only. For instance, Message13 has two features belonging to Class1 and one feature belonging to Class2. In Move message operation, the algorithm will recommend moving this message from Class4 to Class1. In Extract Message operation, the algorithm will recommend creating a new message belonging to Class2 containing the features of Class2. We will name this message as Message13.1 to show that it is a new message extracted from the Message13. Table 21 shows the results:

Table 20: Results of running KSA for Extract Message Refactoring

No.	Point	Representation	KSA Recommendation
1	24	Message 7,Class2	Keep Message 7 Create 7.1 to Class1
2	52	Message14,Class4	Move Message 14 to Class3

			Create Message 14.1 to Class1
3	8	Message3,Class1	Keep it
4	8	Message3,Class1	Keep it
5	48	Message13,Class4	Move Message 13 to Class1 Create Message 13.1 to Class2
6	4	Message2,Class1	Keep it.
7	59	Message15, Class4	Keep it.
8	12	Message4, Class1	Keep it.
9	8	Message3,Class1	Keep it.
10	8	Message3,Class1	Keep it.
11	46	Message12,Class3	Keep it.

Parameter	Value			
			Recall	40%
Initial Temperature	100000000000.0		Precision	40%
Cooling Factor	temperature /10		Ratio	11/2 =5.5
Threshold	Temperature > 1			

6.5 Comparison with GALE:

In this section we compare our KSA algorithm with a recent algorithm named GALE [156]. GALE is a multi-objective evolutionary algorithm that is based on active learning to obtain the Pareto front points. GALE is composed of two main components: one is the division of data and the second is the optimizing process, which is similar to other multi-objective evolutionary techniques. In dividing the data, GALE does not use clustering,

instead it uses a WHERE tool that takes the two poles of each data split; GALE divides the data into various splits and takes the two extreme points (named poles) east and west of each split and evaluates them in each generation of the evolutionary algorithm. GALE's strong point is in less space exploration since only two points of each split are evaluated. Nevertheless, GALE sorts the data before each recursive division of data. An interesting feature of GALE is that it directs the search towards one pole (the better). Table 22 shows the comparison between GALE and KSA algorithms.

Table 21: A comparison between GALE and KSA

Feature	KSA	GALE
Underling metaheuristic	Based on a Single Solution Simulated Annealing	Based on a Population-based Evolutionary Algorithm
What clustering is used?	Kmeans clustering	WHERE clustering
Clustering returned points?	Return the center of the cluster	Return the east and west of the cluster
Number of clusters	Determined by the user (K)	Recursively applied using the median between east and west until a termination point
Exploration	Random Exploration based on the k points returned by the Kmeans	Random exploration towards the preferred boles returned by WHERE tool.
No of evaluation in each iteration (generation)	n	2 points (poles)
Implementation Language	Java	Python

6.6 Large Case Study

In this experiment, we used a large case study in order to see the effectivenss of a search-based algorithm namely: SA in refactoring sequence diagram. The anti-pattern of this case study is depicted in Figure 28 and the refactored diagram is depicted in Figuer 29. Figure 28 shows that Bankserver class should communicate with InterestRate class to compute one function. It would be better if the function setIntestRate is inserted in the bankServer class and thus the class will reference its message. This operation will reduce

the number of classes and will reduce the communication between different classes and increase the communication of loop messages which indicates high cohesion.

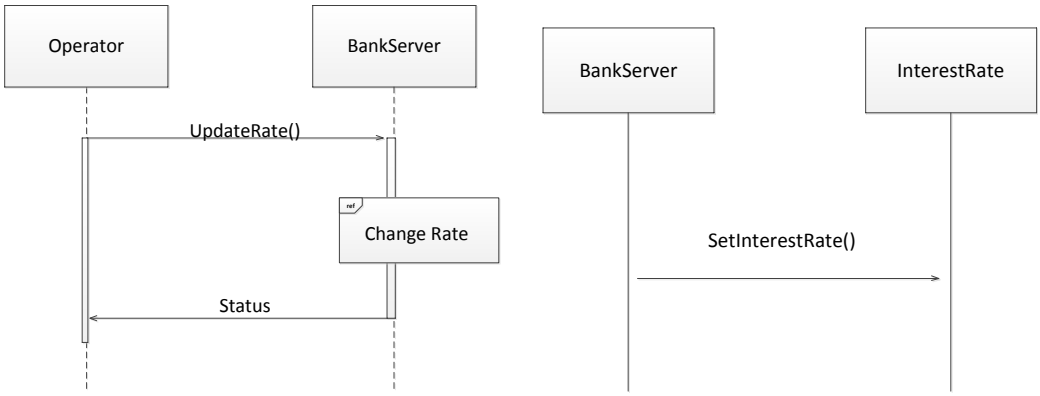

Figure 28: Original Sequence Diagram

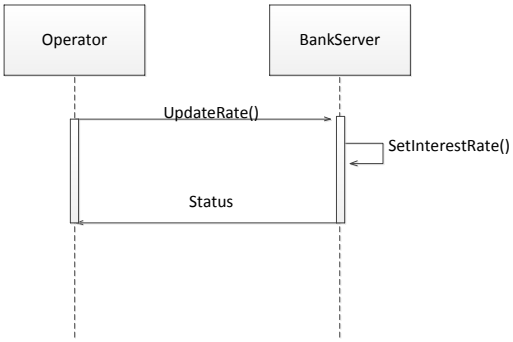

Figure 29:Refactored Sequence Diagram

Using search-based for large case study, we applied SA to detect a sequence anti-pattern in a large case study. We ran it for 25 times and recorded the number of instances detected in each run. Then we calculated the average and the standard deviation of these runs as shown in Table 22.

Table 22: Number of anti-patterns detected in each run of SA

No	Number of instances detected
1	636
2	645
3	648
4	585
5	664
6	707
7	648
8	701
9	575
10	634
11	680
12	612
13	656
14	686
15	664
16	703
17	696
18	679
19	633
20	647
21	667
22	636
23	663
24	655
25	702
Average	656.88
Standard Deviation	34.47018
Ratio of Average anti-pattern detection	656.88/1000= 66%

The case study contains ten thousand messages with a thousand anti-pattern instances. We set the number of non-optimizing iterations of SA to 997. So the number of iterations the SA goes in each run can be found by adding the number of detection to 997. For example, in the first run, SA goes for 636 + 997 = 1633 iteration.

In this experiment, we are not able to list all refactoring suggestions of the algorithm due to space limitations. For example, in the first run: there are 636 refactoring recommendations found by the algorithm. Since the objective of this experiment is to show the performance of SA in a large case study and since SA is working randomly, we calculated the average and the standard deviation of around 25 runs. Table 12 shows the details of this experiment. SA in average was able to find 66% of all instances in the case study.

6.7 Threats to Validity

There are a number of threats that affect the generalization of the results of this research. We ran our experiments on one case study. Other case studies with different characteristics may provide different results. However, we have adopted this case study from the literature in order to ensure the validity of our hybridized approach on a published case study. Another possible threat is that all algorithms are based on random initializations. This might lead to different results in each run. However, since our objective is to show the gain that can be acquired from the hybridization of data mining and search-based algorithms, we can assume that regardless of the initialized points, SA always will gain benefits from the results returned by the Kmeans algorithm.

The proposed hybridized algorithm implements a one-way communication from the Kmeans to the SA algorithm. This can aid the SA whenever it starts and can still have significance on the performance for small problems. However, for larger problems where SA runs for many iterations that modify the elements of the Kmeans clusters, there is no way for SA to seek help from the Kmeans algorithm to cluster the points again. For such

problems, a full integration of the two algorithms with many levels of communication will be very helpful. This can be accomplished in future research.

In a larger problem, where there are many points that do not contribute to the results, clustering algorithms can be very helpful in passing only the good points to the SA algorithm. In our case, the maximum similarity is 2 and the minimum similarity is 1. The range is very short. However, in other problems, the range can span over a wide range between the maximum and the minimum similarity of features. The Kmeans algorithm can divide this range into different clusters, where SA can start from one cluster and move to a neighbor in another cluster. In addition, in a very large space, the resulting cluster from the Kmeans can be huge as well, with points that have different values resulting in a minor improvement to the SA algorithm.

Cohesion and coupling do not capture all aspects of software quality; however, since our objective is to show the effectiveness of the proposed algorithm using multiple competing metrics. Coupling and cohesion metrics are two competing metrics that have been used previously in other research and found to be effective in such context.

Search-based algorithms applied to software engineering introduce some construct validity as outlined by [157]. Construct validity is concerned whether the used measurements are relevant and meaningful to the study. One of these threats is the validity of the cost of executing the fitness function. Although, in our paper, we explained that KSA is able to reach to an optimal state of refactoring using 21 iterations, it is not clear how costly these iterations are with respect of time and resources.

Another possible threat is that we used software quality metrics along with precision and recall metrics to assess the effectiveness of the approach. We reason that KSA is better than SA due to the quality improvement measured by these metrics. However, these set of metrics have been previously used in many studies and have been shown to be relevant for such goal.

6.8 Conclusion and Future Work

Data mining algorithms and search-based algorithms use different approaches to solve optimization problems. Clustering algorithms run based on the similarity between data, while search-based algorithms search the problem space using a guided fitness function. Both of these algorithms suffer from several limitations. In this chapter, we have shown how two instances of these algorithms can communicate with each other to overcome the limitations and produce better results. The application of these algorithms has been tested on a sequence diagram refactoring case study. The experiments show the advantages that can be gained by hybridizing data mining and search-based algorithms on a software engineering domain problem.

Three experiments were performed to show the results returned by the KSA algorithm. The KSA algorithm, after 21 iterations, was able to recommend all the necessary refactoring operations in order to reduce the LCOM value, thus increasing cohesion and reducing the coupling. In terms of recall and precision, if we allow the algorithm to run longer, going through more iterations, it is evident that it is going to retrieve or recall more recommended operations. However, it is interesting to note that the algorithm does not give any incorrect refactoring operation. In all three runs of the algorithm, precision is always equal to the recall value, implying that there is no false recommendation.

We compared our results in the case study with another publication. As they used a different coupling metric, a full comparison could not be made, yet their LCOM value decreased by 2 for one class while our algorithm decreases the LCOM value by 3.

Hybridizing SA with clustering shows interesting results. However, we plan to apply different hybridization algorithms with various software quality metrics. In addition, it would be interesting to observe how the algorithm can scale up with large datasets. In our dataset, the algorithm needs 21 iterations which is arguably acceptable. But testing the algorithm on datasets that contain hundreds or thousands of messages might reveal interesting observations. Nevertheless, we are leaving this to further research.

Another future direction is to use other software metrics with advanced data mining algorithms and search-based algorithms.

CHAPTER 7

CLASS DIAGRAM REFACTORING

There are three major steps done in regard to the class diagram. We are going to list these steps and explain them briefly in the following paragraphs:

- Extracting metrics from classes.
- Extracting smells from classes.
- Set up and run experiments of different algorithms to show their performance.

7.1 Datasets

We will use two open source projects: Eclipse 3.6 and Android 2.3.1 as our datasets. We used Borland Together to extract all class smells from both projects. Together extracts class smells from software automatically based on specified metric values. We have used Together as our smell advisor for two reasons: The manual extracting of smells can take a longer time and they are error-prone. Second, it has been used by other researchers [119].

7.2 Software Metrics

We have extracted 10 metrics from each Eclipse and Android class. Below is the description of these select metrics:

- V (G) (McCabe Cyclamate complexity): measures the complexity of the program which is represented by a control flow diagram. Then the complexity is the number of edges minus by the number of nodes -2.
- DIT (Depth of Inheritance Tree): measures the number of level of inheritance in a class.
- NOA (Number of Attributes per Class): measures the number of attributes in each class
- NLM (Number of Local Methods): measures the number of local methods
- WMC (Weighted Methods per Class): measures the number of methods in each class.
- RFC (Response for Class): measures the number of methods invoked due to a message sent to the class.
- DAC (Data Abstraction Coupling) is the number of abstract data types defined in a class.
- CBO (Coupling between Objects) measures the number of classes coupled with a particular class.
- LCOM (Lack of Cohesion of Methods) measures the number of methods that is not related to any other class through attributes sharing.

- LOC (Lines of Code) measures the number of lines in a code.

These metrics have been selected as they measure different aspects of object oriented software such as: size, complexity, coupling and cohesion.

7.3 Class Smells

There are different class smells indicated in the literature. In this study, we selected four different class smells. The candidate smells are: Data Class, Data Clump, Feature Envy and God Method.

These smells belong to two categories: smells that have been experimented in the literature and smells are not experimented before. We selected smells from the first category since they are of interest to researchers in the domain and thus we want to apply our algorithms on them. In addition, we selected new smells that have not been experimented before, in order to show that our algorithms are applicable to detect these smells with good accuracy. The four smells here are a combination of class and method smells common in the class diagram of object oriented software.

The selected bad smells are:

Data Class: It refers to a class that has data members only and no methods are inside. This is a violation to the object-oriented concept that data and methods should be contained in one entity.

Data Clump: This smell occurs when a class has different data items that are usually passed or processed together.

Feature Envy: This smell occurs when some methods are communicating with other classes more than the one that they are contained in. This smell can result in high coupling and low cohesion. Since the methods are communicating with other classes.

God Method: Similar to God Class, God method is a method that is too big doing so many functions or processes.

The number and types of some of the class smells considered in this paper is in Table 23. It should be noted that some classes might have more instances of the bad smell. However, in our experiment, we do not consider the quantity of bad smells, but the existence of a bad smell in a particular class.

Table 23: Number of class smells in the case study

Class Smell	Frequency in Eclipse	Frequency in Android
Data Class (DC)	3	1
Data Clump (DCL)	4	4
Feature Envy (FE)	2	1
God Method (GM)	8	2

7.4 Experiment Setup

We collected the case study classes data and extracted metric values out of these classes. Then we run Together to automatically identify the number of a particular bad smell in each class. We gathered this data in one excel sheet and fed it to DTREG tool for analysis. DTREG has many built-in algorithms with various parameters that can be tuned up. We repeat the same process for all algorithms, bad smells and case studies. We run our experiments on an Intel I3 processor with 3.30 GHZ and 4.00 GB of Memory running a Windows 7 operating System.

7.5 Algorithm Parameters

We tried different parameters for each algorithm until we reached satisfied results. The set of parameters for each algorithm is illustrated in tables 24-26. The same parameters were repeated in all experiments.

The parameters are usually the same for each algorithm in each experiment. These parameters are the set-ups of the algorithms and are not changed during the learning process. An example of this parameter is the kernel parameter of SVM or the number of layers of MLP network. The full details of these parameters in each dataset for each class smell are given below:

Table 24: MLP parameters

Parameter	Value
Number of Network Layers	4
Number of Hidden Layers	2
Minimum of Neurons in the optimizing Hidden Layer	2
Maximum of Neurons in the optimizing Hidden Layer	20
Cross validation for the optimizing hidden layer	4 folds

Table 25: GEP parameters

Parameter	Value
Population Size	1000
Maximum Generation	2000
Generation without Improvement	1000
Mutation Rate	0.044
Crossover two-point rate	0.3

Table 26: SVM parameters

Parameter	Value
Type of SVM Model	Epsilon SVR
Stopping Criteria	0.00100
Tolerance	1e-008
Range of searching for C parameter	0.1 - 50000
Range of searching for Gamma parameter	0.001 - 20
Range of searching for P parameter	0.0001 - 100

7.6 Cross Validation

Each model or algorithm uses portion of the data for training and then apply it in validation phase. We used a 10-vold cross validation in all models. 10-vold cross validation partitions the data into ten portions, where nine of those portions were used as training set and the tenth one is used for validation purpose. The same process is repeated rotating and shuffling the validation set in each turn.

Cross validation is used to measure the performance of the machine learning algorithms. K-fold can be different in one experiment to another depending on the size of the data. In this chapter, we used 10-folds as it is the one is done by many previous studies in the domain for validation [111, 127].

7.7 Error Measure

There are different error measures used by researchers to evaluate the accuracy of each model. We use two measures: Mean Squared Error (MSE) and Mean Absolute Error (MAE). MSE measures the error or the deviation of the model by taking the average of

the squared difference between the real value (y) and the estimated value (y') as in equation 1.

$$MSE = \frac{\sum_{i=1}^{n}(y-y')^2}{n} \quad (1)$$

MAE measures the error of the model by taking the average of the absolute difference between the real and the estimated value.

$$MAE = \frac{\sum_{i=1}^{n}|y-y'|}{n} \quad (2)$$

These two measures have been used to evaluate the detection of machine learning algorithms [158, 159].

7.8 Results

In this section, we present the results on the experiments. The experiments are intended to answer the following three research questions:

RQ1: Is there a link of algorithms performance in training and validating phases?

RQ2: Is there a difference in algorithms performance when presented with different data sets?

RQ3: Do the algorithms vary in performance when we vary the detection measure?

The purpose of the first experiment is to compare the accuracy of the used algorithms and determine if there is a link between the algorithms' performance in the training and validating phases. In the second experiment we aim to answer RQ2 to compare the performance of each algorithm using two datasets in the validation phase only and

determine if the algorithm shows a pattern in both datasets. The third experiment which is designed to answer the third research question compares the performance of the two algorithms using the two error measures.

Experiment 1: Algorithm performance in training vs validation phases

The objective of this experiment is related to RQ1. So, for each class smell for the two data sets, we compare the performance between training and validation. We recorded our observations and analyses the output behavior.

Table 27: Algorithm performance in Training vs validation using MSE

Smells / Algo.	DC		DCl		FE		GM	
	Training	Validation	Training	Validation	Training	Validation	Training	Validation
MLP	0.01	0.09	0.04	0.05	0.02	0.02	0.03	0.11
GEP	0.02	0.05	0.03	0.05	0.01	0.04	0.01	0.08
SVM	0.03	0.03	0.02	0.04	0.02	0.02	0.05	0.07

Table 28: Algorithm performance in Training vs Validation using MAE

Smells / Algo.	DC		DCl		FE		GM	
	Traini	Validati	Traini	Validati	Traini	Validati	Traini	Validati

	ng	on	ng	on	ng	on	ng	on
MLP	0.01	0.00	0.04	0.07	0.01	0.02	0.01	0.03
GEP	0.01	0.02	0.03	0.04	0.00	0.02	0.01	0.03
SVM	0.01	0.01	0.01	0.03	0.01	0.01	0.00	0.03

In our analysis, we are going to start with an observation of the performance of each algorithm in Training vs. Validation phase. Discussing the algorithm accuracy in training vs. validation phase can unleash many interesting observations; that is helpful to understand the scalability and the generalizability of the algorithm.

In machine learning domain, there are two concepts that lead algorithms to give a misleading performance to non-experts. These are known as "over-fitting" and "under-fitting". Over-fitting occurs when the algorithm tries to build a model that is so accurate to the training data. In this scenario, the model fits the training data very well, but it fails drastically when it is fed by a new point. Regardless of the algorithm superiority in the training phase which may mislead the designer, it could be totally useless when it is presented with new data. Under-fitting is just the opposite concept of over-fitting. In under-fitting, the algorithm cannot even perform well in the training phase and this can reveal an issue in the number of data fed to the algorithm. In another word, under-fitting alerts the designer that the data is so small that the algorithm cannot build any model with acceptable accuracy to fit the training data. One reason to account for that could be the complex relation between data.

Eclipse Dataset

Figure 28 shows a comparison between algorithms' performance in the training vs. validation phases for the four smells in Eclipse. Figure 28.a shows that MLP performed very well in the training phase. The penalty here is the over-fitting symptom which leads to a drastic bad performance in the validation phase. SVM shows an interesting observation of having an identical performance in training and validation. This gives us a hint on the way SVM can build its model in a way not to over-fit and gives an accurate detection. Figure 28.b shows that SVM performed the best in the training phase. Nevertheless, this does not indicate an over-fitting as other algorithms; but it continues to obtain the best validation performance. In Figure 28.c, GEP showed the best performance in the training phase. Like MLP, it suffers from over-fitting and performed very bad in the validation phase. Figure 28.d shows over-fitting symptoms of both MLP and GEP performing very well in training and drastically bad in validation. SVM on the other hand build a model that gives an adequate variation between training and validation in all the experiments using Eclipse dataset.

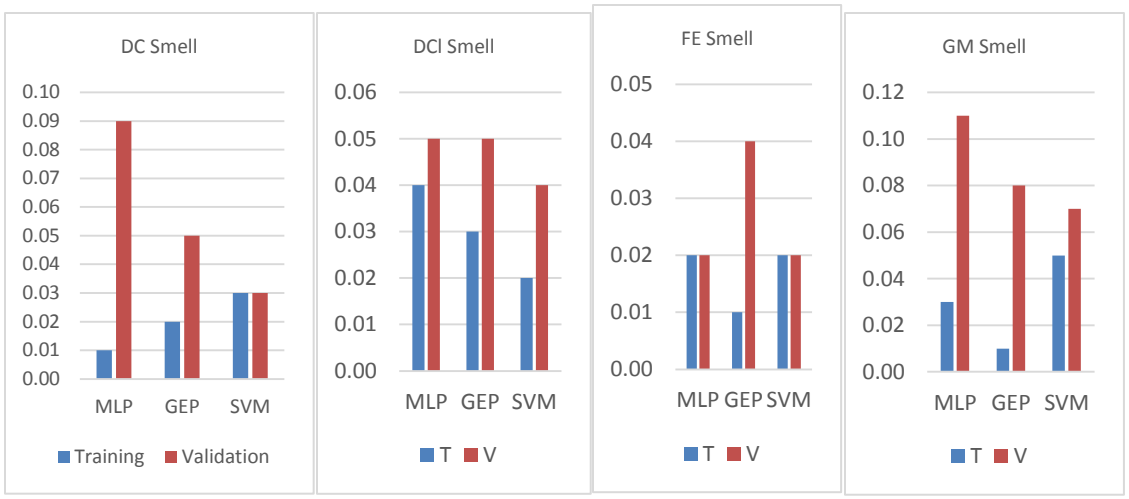

Figure 30: Algorithms' performance in training vs validation phases in Eclipse.

Figure 29 show a comparison between algorithms' performance in the training vs validation phases for the four smells in Android. Figure 29.a, shows that the three algorithms performed similarly on the training phase when detecting DC smell. However, GEP suffered from an over-fitting symptom and its performance is very bad. SVM shows an identical performance in training and validation. MLP surprisingly performed exceptionally very accurate. There is something under the hood that derived MLP to get such an outstanding result. In figure 29.b, SVM performed the best in training phase but that does not lead to over-fitting since it obtained the best result in the validation as well. Figure 29.c shows another instance of an over-fitting symptom. GEP which performed incredibly well in the training phase, failed in the validation phase. SVM and MLP performed similarly in the training phase, but SVM keeps its performance in the validation phase while MLP performed like GEP. Figure 29.d shows that the three algorithms performed similarly to each other in the training and the validation phase.

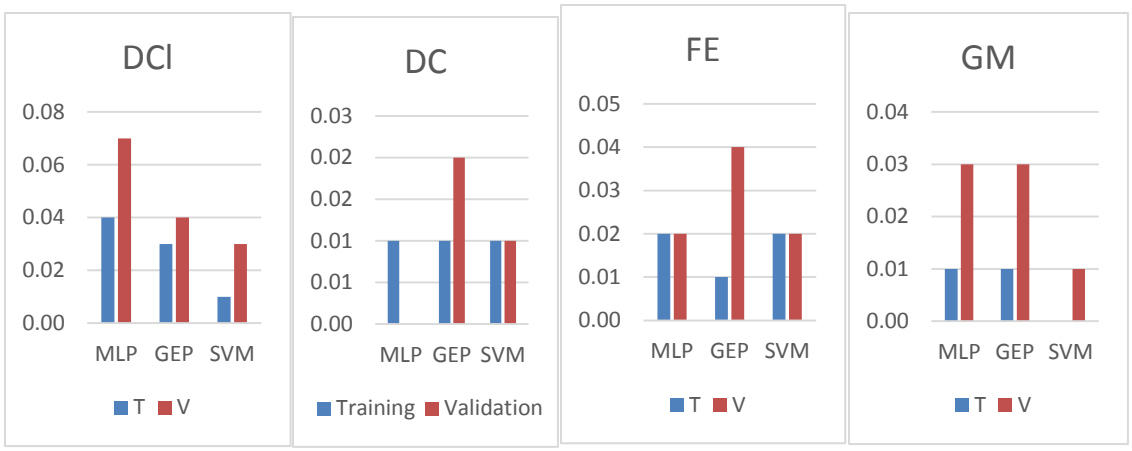

Figure 31: Algorithms' performance in training vs. validation phases in Android.

7.8.1 Experiment 2: The algorithm performance on two data sets

The objective of this experiment is to show which algorithm performs well in each data set.

Figure 30 shows the performance of the three algorithms in the training phase using MSE and MAE measures. Figure 30.a and 30.b show the performance on Eclipse dataset. The results show a pattern in algorithm performance when MSE or MAE is applied. So regardless of the error measure, the performance might differ, but they follow a pattern. For example, we can see that SVM suffers the worst performance in training using MSE in detecting GM smells. In MAE measure, though the value is different, they perform the worst again among the three algorithms. Detecting FE smells, we see that GEP performs the worst and MLP and SVM perform similarly using MSE measure. Looking at MAE measure chart, we see that the values are different but the same ranking is observed.

Figure 30.c and 30.d show the performance of these algorithms in the training phase using MSE and MAE measure applied to Android dataset. Here, we see that SVM performs closely using either measure across the four data smells. MLP and GEP however behave unexpectedly in these figures. MLP performs closely to the others in detecting GM using MSE measure. However, it performs drastically bad as compared to the others when it is applied to detect GM using MAE measure. The same applied to GEP when detecting FE, for example. It performed the best in the training phase using MSE error measure, while it performed the worst when MAE is applied.

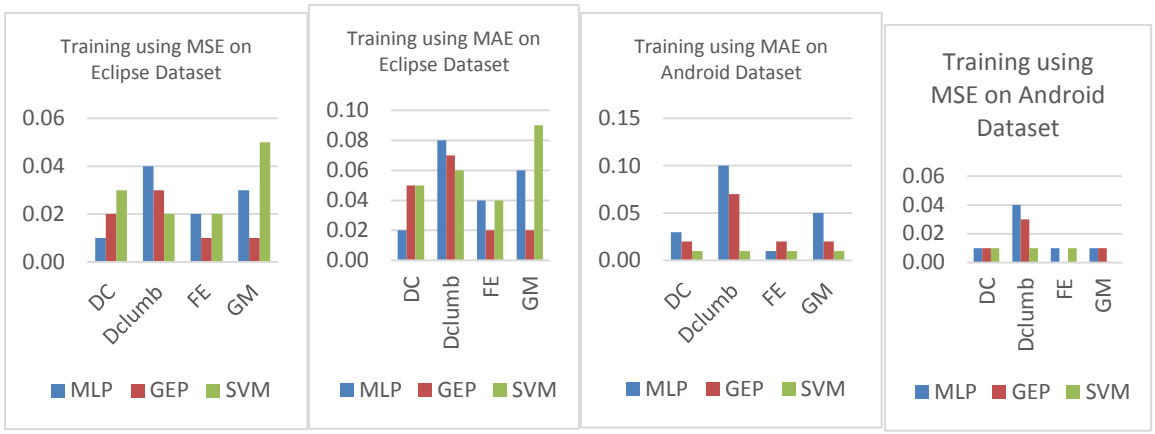

Figure 32: Algorithms' performance using two data sets in training phase

Figure 31 shows the performance in the validation phase. This phase is the phase that most researchers are interested in. Figure 31.a and 31.b show the results across the four data smells using the two measures applied to Eclipse dataset. We chose a line chart to illustrate clearly the superiority of one algorithm over others if it exists. The figure shows that SVM performs the best for all data smells using the two measures. This is an interesting observation indicating the superiority of SVM. MLP and GEP perform unexpectedly in these two measures and no observation can be concluded in that regard. Nevertheless, it seems that all algorithms perform closely to each other using MAE measure with SVM still is the best.

Figure 31.c and 31.c show the algorithms' performance when applied to detect the four bad smells using the two error measures on Android dataset. Again, SVM shows the superiority in almost detecting the four bad smells using MSE and MAE measures. We said almost because it is only in detecting DC using MSE measure, MLP seems to perform the best. However, we should be very cautious because MLP performed the worst when applying MAE measure. Unlike Eclipse dataset, there is a gap in performance between SVM and other algorithms using either error measure (Except in the DC instance where MLP performed surprisingly the best using MSE).

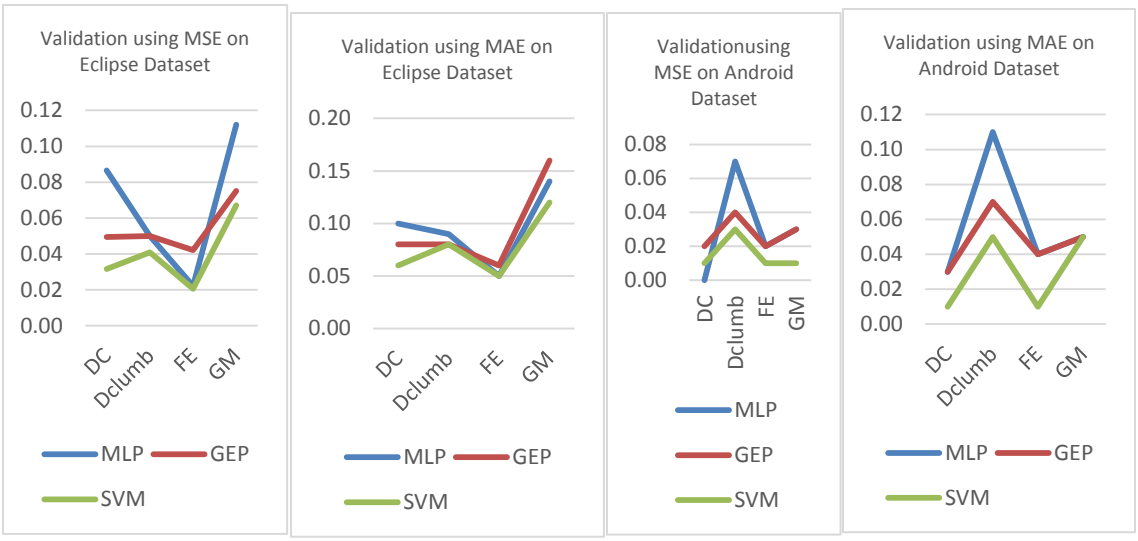

Figure 33: Algorithms' performance using two data sets in validation

7.8.2 Experiment 3: The contribution of different error measures on the algorithm performance

The objective of this experiment is to show the algorithms' performance using two measures MSE and MAE.

Figure 32.a and 32.b show the algorithms' performance in both datasets side by side when applied to detect DC smell using MSE and MAE measures. GEP and SVM show a pattern in both of these figures. MLP performance is not consistent. It performed the worst in Eclipse and the best in Android using MSE measure. Again, using MAE, MLP performs the worst in Eclipse and competitive to GEP in Android. This can sign the undetectability of MLP when presented with different datasets.

Figure 32.c and 32.D show the algorithms' performance in both datasets side by side when applied to detect DCl smell using MSE and MAE measures. The three algorithms performed closely using Eclipse dataset. However, different measures can boost the performance of some algorithms as we can see that GEP is able to compete with SVM using MAE measure. In Android dataset, there is a clear gap in the performance of the three algorithms using both measures.

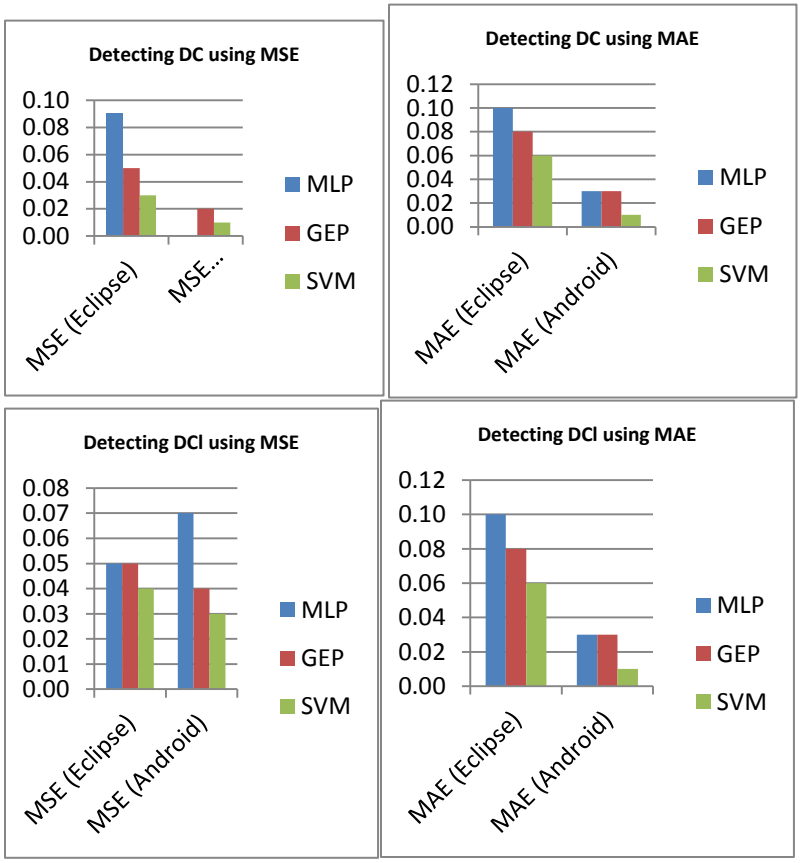

Figure 34: Algorithms' performance using error measures in predicting DC and DCl

Figure 33.a and 33.b show the algorithms' performance in both datasets side by side when applied to detect FE smell using MSE and MAE measures. We can observe that GEP observed better using MAE measure than using MSE measure when applied to Eclipse dataset. This is in accordance with the observation in Figure 28. GEP tries to optimize MSE in the training phase to the limit of over-fitting so it performed better using MAE in the validation phase.

Figure 33.a and 33.b show the algorithms' performance in both datasets side by side when applied to detect GM smell using MSE and MAE measures. SVM performed the best or competitively the best. MLP performed the worst in Eclipse using MSE measure and closely to the best using MAE. This observation is in lie with the above observations that MLP might perform differently using different error measures.

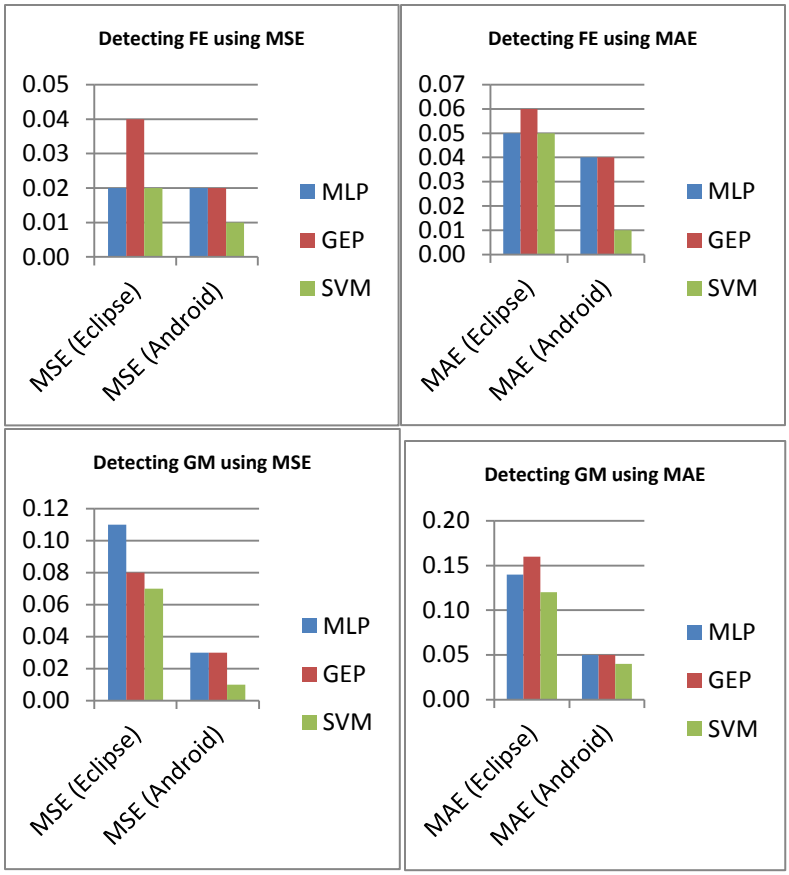

Figure 35: Algorithms' performance using error measures in predicting FE and GM

7.8.3 Discussion of the results

To be able to draw some inference on the observation of the algorithm performance, we applied some statistical measures that is applied by other researchers [127] in analyzing the results. We calculated the mean and the standard deviation of the three algorithms for the two studied datasets as shown in Table 29.

Table 29: The mean and standard deviation of the algorithms

	Eclipse			Android		
Algorithm	Mean	StDev.	CV	Mean	StDev.	CV
MLP	0.07	0.04	0.585199	0.04	0.029	0.769796
GEP	0.05	0.01	0.266019	0.03	0.013	0.405515
SVM	0.04	0.02	0.496658	0.02	0.009	0.441378

The mean of SVM algorithm is the best in Eclipse and Android datasets, followed by GEP and then MLP. We are going to use the coefficient of variation (CV) to determine if the variation is high or not. CV is calculated by dividing the mean by the standard deviation. We inferred that GEP has the lowest variation in Eclipse and Android datasets. This might be attributed to the evolutionary nature of the GEP algorithm and thus in each iteration of this algorithm, it applies a crossover and mutation that reduce the dispersion of the data and thus results in lower standard deviation and sequentially lower CV.

Since the results reveal that SVM performed the best, we would like to determine if this can be statistically significant or not. Using unpaired t-test between the MLP and SVM in the Eclipse dataset reveal that the results are not significant since the t value is 1.2 and the p-value is 0.27 > 0.05 at a 95% confidence interval. t value for GEP vs SVM is 1.1 and the p-value is 0.3 >0.05 which is not significant. In Android dataset, no significance

between SVM and MLP since the t-value is 0.9 and the p-value is 0.3 > 0.05 and no significance between SVM and GEP since the t value is 1.8 and the p-value is 0.12 > 0.05.

For MLP to give good results, the number of hidden layer should be two. Having one hidden layer affects the performance. Having so many hidden layers will make the algorithm take a long of time. In addition, the number of neurons in the second layer should be minimal. Increasing the number of neurons in the hidden layer does not necessarily improve the accuracy. For example, in detecting DC, DCL and GM, we found that having 2 neurons in the hidden layer gives the best result. Increasing the number of neurons will affect the results. In detecting FE, 4 neurons are the optimal. However, setting the number of neurons to two neurons will get a very close results to the 4 number of neurons. In a nutshell, setting the number of hidden layer to 2 and set a few neurons in the hidden layer usually gives a good result.

GEP usually takes longer time to run since we chose a 1000 instance in the population and let it run for 2000 iterations. Having very low population gives a faster result but with low accuracy. To get a good result with low population, the designer needs to increase the mutation rate. When we put the population to 10 and the mutation rate to 0.3, it gives us a very good result for detecting FE in Eclipse. However, when we set the mutation rate very high to 0.5 (which means mutating half of the population), the accuracy dropped.

SVM in the two datasets using two different measures on different data smells performs slightly close. MLP and GEP on the other hand has a tendency to optimize using one

error measure over another. Thus, it is recommended for the designer to test accuracy with a different measure when these algorithms are tested. GEP is an evolutionary algorithm that targets a fitness (objective) function. In our experiment, we set the fitness function's parameter of GEP to MSE, so it is expected that GEP tries to optimize this measure the most.

7.9 Threats to Validity

Though of our extensive work to set up and run experiments that produce good valid results, there are several threats of validity that we should point out: The tuning of the parameters of the three algorithms are based on trial and error. Different setting of some parameters may impact the performance. However, we tried several combinations of parameter settings to reach the results produced in this paper.

We used Together as an advisor of the presence of class smells. This automatic process might not reveal as much as manual process. However, Together tool was used by other researchers to automatically identify class smells.

In addition, we relied on a DTReg tool to run our experiments. Different tools might have different code-optimization methods; other tools that apply the same settings of our algorithms might produce results faster or slower.

We experimented our algorithms on two datasets of object oriented software. Though, this might affect the generality of our results, we studied different data smells in each dataset. This increases the confidence that a particular algorithm performance can be generalizable to all data smells in average. SVM in average performed the best in both of these datasets across the four data smells.

Our results are obtained by training the algorithm using a set of features (metrics) equal to 10. Though we selected metrics that are more related to the data smells in this study, more metrics may produce better results and could affect the training time and the algorithm complexity.

The two studied datasets are open source datasets and as such they might not be a good candidate of representing the systems that used in industry as pointed out by Wright et al. [160].

7.10 Conclusion and Future Work

Bad smells can be a good indicator of class errors as revealed from the literature. Machine learning algorithm can be applied successfully to detect different types of class smells. In this chapter, we applied three different machine learning algorithms to empirically investigate which of this algorithm gives a good detection accuracy of four different class smells for two large open source software. SVM shows a promising result in obtaining the best detection accuracy comparing to other algorithms under study. In addition, SVM balanced the results in the training and the validation phase, so there is no high difference of accuracy when running SVM in training vs. validation. GEP takes long time to build a model since it is an evolutionary algorithm. It has the best coefficient of variance among the three algorithms. MLP even though performed the worst among other algorithms, this low performance is not statistically significant in comparison to SVM at 95% or 90% significant level.

The results show the effectiveness of SVM in building detection model and the validity of the results generated in the training model. This has a good practical implication for

any designer who has few training data and it assures that SVM will perform closely on the real validated data. In addition, GEP algorithm coefficient variance have a practical implication that GEP will build a model that obtain a closer result to the mean.

Our future work should be dedicated to the differences in results among different error measurements. In addition, it is helpful to compare the performance of algorithms in training and validation phases and how can that be formulated into general rules. This will help researchers with few trained data of making strong judgment of their algorithm performance when it is applied in real context.

CHAPTER 8

MULTIPLE-VIEW DIAGRAM REFACTORING

The multiple-View model proposed by Misbhauddin [16] combines three diagrams: class diagram, sequence diagram and use case diagrams. Each diagram represented one view as illustrated in Figure 1. Misbhauddin extended the metamodel of these three diagrams to form one model named an integrated model. Then, he ran some experiments to show that the integrated model can reveal some hidden smells of each diagram that was not clear when refactoring each diagram individually.

8.1 Motivation and Objective

Each UML diagram has a different set of smells or anti-patterns that alerts the designer to a potential degradation of the design quality. However, some smells are not that obvious and they cannot be detected by focusing on one diagram. As such, a manual inspection by experts looking at different diagrams belonging to different views can spot some of these smells. Thus, an integrated UML view that combines three diagrams representing different views was proposed by Misbhauddin. In his work, he asserted that an integrating UML diagram will expose some smells that was not be able to be detected from a single diagram alone.

In this work, we are going to use one of the design smells that he used and provide an automatic method of detection and refactoring using search-based algorithms. It is

expected that refactoring a smell that is connected to more than one diagram may have different impacts on each diagram. Search-based algorithm can be handy here in automating the detection and refactoring of these smells considering a positive impact on the three diagrams using different quality metrics that is related to each diagram. The process is fully automatic and transparent to the users. One merit of using search-based algorithm is the ability to instruct the algorithms to detect and refactor smells that impacts positively on the three diagrams and ignores other smells that might impact negatively on some diagrams.

Moreover, integrated or multiple-view diagram tends to be lengthy and large. In this work, we consider combining three diagrams only following Misbhauddin's work. In that regard, search-based algorithms can be handy in searching a multiple-view diagram that usually contains tens or hundreds of components. Different search-based algorithms have different capabilities of performance. In this work, we are going to compare three search-based algorithms: Hill Climbing (HC), Late Acceptance Hill Climbing (LAHC) [161] and Simulated Annealing (SA) [148].

The reason behind selecting these algorithms is the lack of studies that use search-based algorithm to refactor multiple-view diagram. Thus, HC is a good candidate to show the applicability of applying search-based algorithms to this problem since it is considered the easiest algorithm that can be understood by any user. LAHC is more efficient that HC but it still simple and fast making it a good competitor to HC and thus it is included. SA algorithm is used by many authors in refactoring software code [162, 163] and its inclusion to see its strength in refactoring multiple-view design model is expected.

The objective of this study is to answer the following two research questions:

RQ1: How the different algorithms can detect and refactor a smell in a multiple-view diagram?

RQ2: How the detection of a smell in multiple-view model is different than the detection of the smell in a single-view diagram?

8.2 Research methodology

We are going to refactor a multiple-view diagram composing of usecase diagram, sequence diagram and class diagram. This refactoring involves two steps: detection of an anti-pattern and correcting the diagrams. We have selected one anti-pattern named: Creeping Featurism that usually exist in use case diagram. Creeping featurism is a type of functional decomposition where the designer would like to split a large function into smaller functions where each function is doing a specific job. This is an interesting concept since it promotes modularity. The issue is that when the designer excessively decomposes a function into very small functions that almost do nothing and subsequently increases the complexity, the size of the diagram and the interaction between these functions. As El-Attar and Miller [164] mentioned this is a use case anti pattern. He and Khan even provided a semi-automatic approach using graph transformation [165]. They specified the conditions of this smell in an inclusion relationship. However, by looking at the usecase diagram only, the refactoring tool might falsely reports any inclusion as an anti-pattern. By allowing the algorithm to look at the sequence and the class diagram, the algorithm will have much information assisting it in determining the anti-pattern correctly. The pseudo code of checking and correcting the diagram is provided below.

We used three search-based algorithms to detect this anti-pattern in the above mentioned diagrams and refactor it to have a good reflective impact on the quality of these diagrams using a predefined set of quality metrics. Use case metrics are: Number of use cases, Number of Actors and a complexity metric as proposed by [144, 166] . For sequence diagram, we are going to use the number of lifelines, the number of messages, the number of direct messages and the number of self-messages. For a class diagram, we measured the number of classes, the number of attributes and the number of operations in a class. The process of detecting and refactoring in the multiple-view diagram is illustrated in figure 34.

For a use case diagram, we are going to use a set of metrics that captures complexity and the size quality of the diagram since use case diagram depicts the system functionality. Since sequence diagram is more about interaction between different lifelines using messages, so it is of interest to measure the decrease of increase of this interaction before and after refactoring. Class diagrams complexity is usually measured by the number of its attributes and methods and as such calculating the effect of refactoring on these metrics is interesting.

First we will acquire use case, sequence diagram and class diagrams as input. Then, we will encode these diagrams to facilitate applying our search-based algorithm. We encoded these diagrams as objects using java language. We presented the encoding function to the three different algorithms. We followed Misbhauddin's condition for detecting the anti-pattern in a multiple-view diagram. The algorithm then attempts to correct the anti-pattern and automatically calculate the quality metrics after refactoring and provided an encoding version of the refactored diagrams.

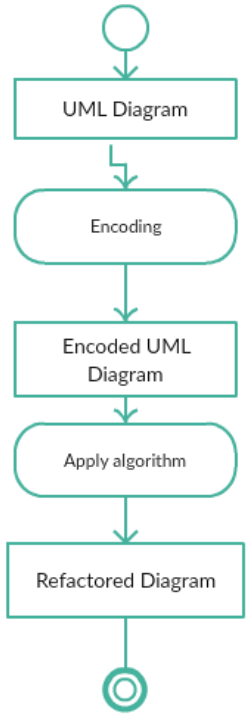

Figure 36: Multiple-View diagram refactoring process.

8.3 Dataset

We used an adopted published case study [167]. This case study has one use case diagram. We have created corresponding sequence diagrams and class diagrams. Then, we have encoded them . Figure 35 shows the original diagram of the case study.

We are going to use one design smell popular in usecase diagram known as Creeping Featurism. Creeping Featurism is a smell that can be detected in multiple view models and refactoring it will have an impact on the different components of the multiple-view model.

We have created the sequence diagrams and the class diagrams of the above use case study accordingly. In addition, we have plugged 9 smells of the creeping featurism smell. Thus, the total number of creeping featurism smell is 10.

We are going to test three search-based algorithms on this case study. Here are some difficulties that this case study imposes on the algorithms:

The encoded case study using java language included three diagrams sequence, class and use case diagrams, but the smell exists in the use case initially. So the algorithm must search and locate use cases in order to be able to detect a possible smell.

There is no connection between the existence of one instance of this smell and another instance. So, the algorithm might be able to locate one instance but it may fail to continue finding another one. There is no clear connection between two instances of the smell and thus a neighborhood function of the algorithm is hard to be defined.

The number of the smells is fewer than the number of use cases in the diagram; consequently, it is fewer than the number of components in the model. So out of more than 120 components in the model, the algorithm should detect the 10 smells. This corresponds to less than 10% of the search space.

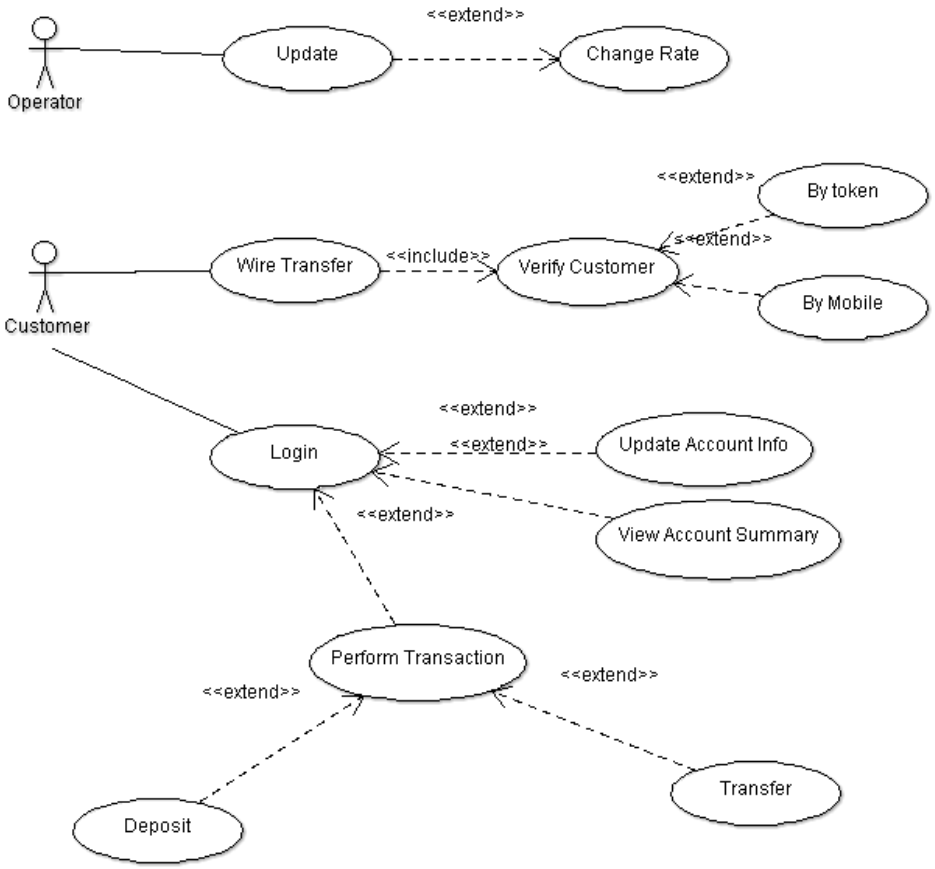

Figure 37:The original Case study

8.4 Experiment Setup

We have run our algorithms using Windows 7 on a computer that is running Intel pentieum I3 with 3.30 GHz and 4 GB of Memory. HC is parameterless, so there is no tuning of this algorithm occurs. LAHC uses a memory list guided by the following formula:

$$s = I \bmod L$$

where s: represents the selected point

I: is the iteration number of the algorithm

L: is the length of the memory list

SA initial temperature is 1000000000.0 and the cooling of the temperature is: temperature /100.

8.4.1 Detection Phase

In the detection phase, the algorithm selects a random number (point) to start with. Then, the algorithm checks if this points corresponds to an anti-pattern or not. The specified anti-pattern has some conditions that the algorithm searches for. If a match is found, this means the anti-pattern is detected. In this paper, since we are dealing with a three diagrams, the anti-pattern must be checked in the three diagrams.

8.4.2 Refactoring Phase

After the algorithm detects an anti-pattern in the detection phase, the algorithm corrects the design defect using a refactoring operation. The refactoring operation is applied to the three diagrams: use case, sequence and class diagram. After correcting the design defect, the algorithm calculates the complexity metric of the usecase. If the complexity value is reduced, the algorithm continues moving to detect another anti-pattern and calculate the quality metrics of the three diagrams to ensure that any refactoring results in a positive impact on these metrics.

In the following section, we will show only the portion of the diagrams where the design smell is detected and the same diagram after refactoring. We will show for the three diagrams: use case diagram, sequence diagram and class diagram.

Below we show the pseudo code of detection and refactoring of this smell:

```
Begin
        Read diagram
        For each component
                If component == usecase
                If usecase inclusion is 1 and usecase not connected to actor
                Check sequence diagram
                If lifeline of inclusion and included diagram is in the same lifeline
                Check for class diagram
                Is the corresponded classes of the included usecase is a data class
                Then detected and it is a candidate of refactoring

                Remove the included usecase
                Remove the lifeline representing the intersection between including and included usecase
                Add a self-message from the lifeline representing the including usecase to itself
                Remove the data class component
                Add all of its attribute and methods to its parent class
`       End if
        End if
        End for
```

Use case refactoring

In figure 36,The use case “Change Rate” exhibits a creeping featurism smell and the algorithm is going to refactor it by removing the “Change Rate” use case and eliminate the inclusion relationship as in figure 37.

Sequence Refactoring

The following diagrams show the portion of the sequence diagrams corresponded to the use case that has a design smell. We have two sequence diagrams before refactoring as in Figure 37. After refactoring, we have only one sequence diagram and one self-message have been added.

Figure 38 shows the sequence diagram after refactoring. In this diagram we see that a new self-message is introduced in the bank server life line.

Class Refactoring

The refactoring of this diagram in figure 40 will remove the interestRate class completely from the class diagram and the inheirtance sequenentially will be removed too. The attributes and methods of InterestRate class will be moved to the BankServer class. Thus, the number of classes will be reduced as in figure 41.

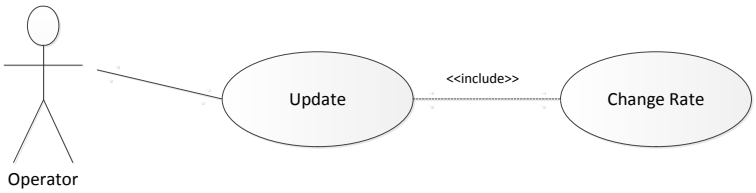

Figure 38:Use case Diagram showing Creeping Featurism

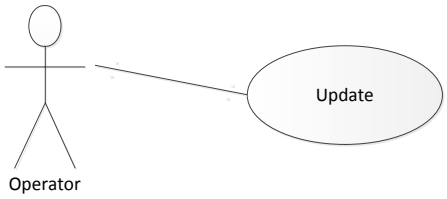

Figure 39:Use case diagram after refactoring

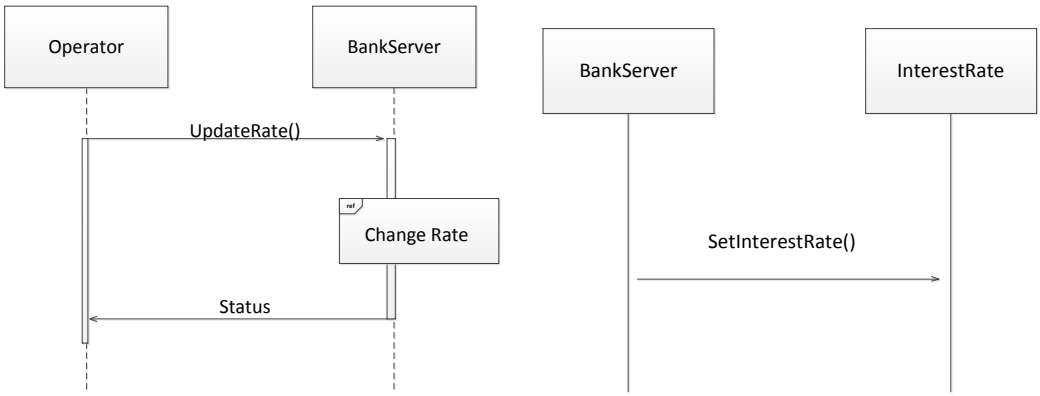

Figure 40: Sequence Diagram Before Refactoring

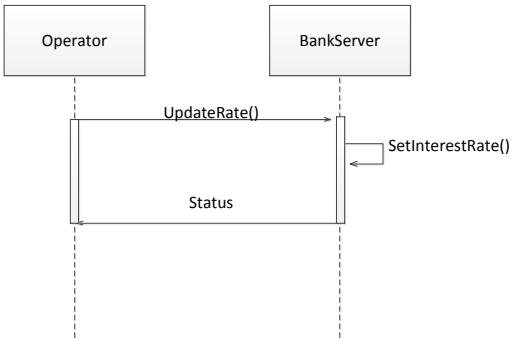

Figure 41: Sequence Diagram after Refactoring

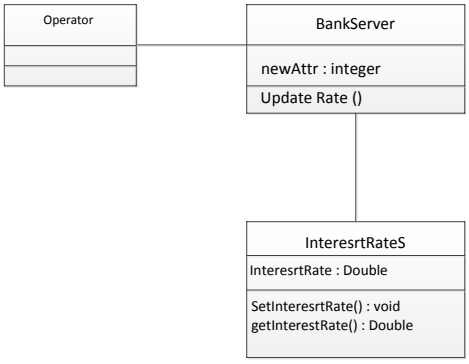

Figure 42: Class Diagram before Refactoring

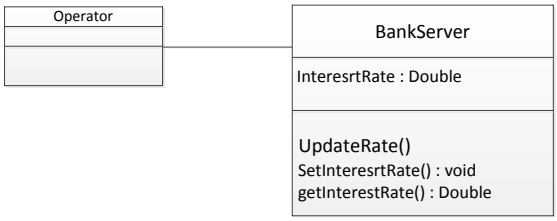

Figure 43: Class Diagram after Refactoring

8.5 Results and Discussion:

In this section, we present the results and discussion on the experiments. First we want to compare between three different search based algorithms in detecting and refactoring multiple-view UML diagram. We realize that since the dataset contains three models and the smell exists mainly in the use case diagram which represents around third of the dataset, it is hard for simple basic search-based algorithm to randomly identify the good starting point. In that regard, we started with HC algorithm since it is very naive. We found that it takes 13 runs for HC to be able to reach to a good starting point. Table 30 shows that number of runs, the starting random point (R) and the impact of the detected use case smell on sequence and class diagram. We found that HC was able to detect the smell when it starts at the point "43". We did not measure the quantity of the impact, we just mentioned whether there is a quality improvement or not. The (=) sign indicates that the metric values of these three diagrams are not changed. The (↑) indicates an increase in quality while (↓) indicates a decrease in quality.

Experiment 1: it is targeting RQ1 and it consists of three sub-experiments:

Experiment 1.1: How many runs does it take HC algorithm to detect a smell in multiple-view diagram? What impact does it have on other diagrams?

In the table, we see that we have performed 13 runs of the HC algorithm. D in the second column referred to detection of the algorithm to the anti-pattern. "r" refers to the random number generated by the algorithm. The last three columns show the impact of each run on the use case, sequence and class diagrams. In the first 12 runs, the algorithm didn't

detect any smell, so the impact is not changed. In the run number "13", the algorithm detects and refactor the smell, and that leads to improvement in the three diagrams as can be seen from table 33.

Table 30: The results of running HC on the dataset

Run	HC	r	Impact on usecase	impact on sequence	impact on class
1	=	8	=	=	=
2	=	118	=	=	=
3	=	86	=	=	=
4	=	85	=	=	=
5	=	109	=	=	=
6	=	63	=	=	=
7	=	66	=	=	=
8	=	27	=	=	=
9	=	79	=	=	=
10	=	105	=	=	=
11	=	1	=	=	=
12	=	22	=	=	=
13	D	43	↑	↑	↑

Experiment 1.2 How many runs does it take LACH algorithm to detect a smell in multiple-view diagram? And how many instances of the smell it is able to detect in one run? What impact does it have on other diagrams?

In this experiment, we see that LAHC is able to find two good starting points "49" and "41" as shown in table 30. We run the algorithm 13 times as we did with the HC. We noticed that from the point "49", the algorithm is able to do 2 good points of refactoring. Starting from the point 41, the algorithm is able to find 6 more refactoring opportunities. Unlike HC where it only finds one smell point, and due to its greedeness design, it failed when the next point was not a smell point, LAHC is able to find other good smell points if it starts from good one. That can be attributed to its memory structure.

Table 31: The results of running LAHC on the dataset

Run	LAHC	r	# of smells	Impact on use case	impact on sequence	impact on class
1	=	79	0	=	=	=
2	=	96	0	=	=	=
3	D	49	2	↑	↑	↑
4	D	41	6	↑	↑	↑
5	=	62	0	=	=	=
6	=	100	0	=	=	=
7	=	105	0	=	=	=
8	=	66	0	=	=	=
9	=	118	0	=	=	=
10	=	4	0	=	=	=
11	=	113	0	=	=	=
12	=	58	0	=	=	=
13	=	53	0	=	=	=

Experiment 1.3: How many runs does it take SA algorithm to detect a smell in multiple-view diagram? And how many iterations does it take the SA before detecting another run? What impact does it have on other diagrams?

Unlike the greedy algorithm HC, SA allows the algorithm to deterioriate for some runs before terminations. The number of runs can be determined by the user. In this experiment, we want to experiment the number of runs that SA can deterioriate before succeeding in finding another refactoring point. We set the threshold to 13 loops. That means, in each run, SA is permitted to deteriorate for a maximum of 13 loops only. If it finds a refactoring point within this threashold, the impact is investigated. If SA took more than 13 loops to find a refactoring point, then it is not counted. As we did with HC

and LAHC, we ran the algorithm 13 times. In the running number, 1,3,4,5,9 and 11 SA is able to find a refactoring point within the threshold (13 loops). Let's drill down in Table 3. In The first run, we see that r equals 27. This is the initial random point started by SA. Then, it took the algorithm 3 loops to be able to find a refactoring point. In the second run, it took the algorithm 19 loops to find a refactoring point. This is greater than our threshold of 13, so we didn't count it as a successful run of the algorithm. So in the total 13 runs, SA was able to succeed 6 times in finding a refactoring operation.

Table 32: The results of applying SA on the dataset

Run	SA	r	# of iterations before the second smell is detected	Impact on use case	impact on sequence	impact on class
1	D	27	3	↑	↑	↑
2	=	109	19	=	=	=
3	D	118	1	↑	↑	↑
4	D	103	3	↑	↑	↑
5	D	18	9	↑	↑	↑
6	=	117	16	=	=	=
7	=	32	16	=	=	=
8	=	66	28	=	=	=
9	D	28	6	↑	↑	↑
10	=	80	22	=	=	=
11	D	2	6	↑	↑	↑
12	=	11	> 30	=	=	=
13	=	76	17	=	=	=

The previous section compares between the three algorithms in their ability to find a refactoring operation and the number of loops it requires for some algorithm such as SA

to find the first refactoring opportunity. In this section, we will focus on validating the impact these algorithms have on the described case study. We will use a metric-based approach since this is the approach used by Misbahuddin to validate the refactoring impact on the UML diagrams. As in Table 30, HC was able to find one refactoring opportunity when it starts at r= 43. This is the only one refactoring opportunity discoved by HC in 13 runs. So we are going to use it here to validate its impact on the three UML diagrams using the metric-based approach.

Table 33: The impact of HC on the metrics of use case, sequence and class diagrams

	Metric	Before	After	Quality Improvement
HC algorithm	# of UC	30	29	↑
	# of Actors	11	11	=
	# of Lifelines	49	48	↑
	# of Messages	49	49	=
	# of Direct Messages	49	48	↑
	# of Self-Message	0	1	↑
	# of Classes	30	29	↑
	Avg. # of Attributes in Class	0.33	0.34	↓
	Avg. # of Operations in Class	1.2	1.2	=
	UC complexity metric	239.81	239.79	↑

LAHC was able to find two refactoring opportunities in the 13 runs. We will explore the second one which is at r=41. LAHC was able to find 6 refactoring operation during its running until termination. We can see in Table 6, the impact on the usecase, sequence and class metrics.

When SA starts from the point “55”, it is able to find 8 refactoring opportunities. Of course the relaxation condition here on the deterioration was high to allow the algorithm to run several loops before termination. If time and effort is afforded, then SA is able to find 8 refactoring opportunities out of 10 which constitute for a recall of 80%. Table 34 shows the impact of this algorithm on the use case, sequence and class metrics.

Table 34: The impact of SA on metrics of use case, sequence and class diagram

	Metric	before	After	Quality Improvement
SA algorithm	# of UC	30	22	↑
	# of Actors	11	11	=
	# of Lifelines	49	41	↑
	# of Messages	49	49	=
	# of Direct Messages	49	41	↑
	# of Self-Message	0	8	↑
	# of Classes	30	22	↑
	Avg. # of Attributes in Class	0.33	0.45	↓
	Avg. # of Operations in Class	1.2	1.63	↓
	Initial Value of UC complexity metric	239.81	239.43	↑

To be able to detect the creeping featurism anti-pattern, the designer must look at the behavioral level specifically to the class diagram. If the included use case is represented in the class diagram as data class with only getter and setter operations, then it the creeping featurism anti-pattern is confirmed. By looking at the use case diagram only, it is not enough to detect the anti-pattern accurately.

We have changed 5 instances of the above anti-pattern. These 5 instances still have an inclusion relationship and satisfy the conditions of creeping featurism anti-pattern at the

use case level. However, the corresponded classes to these use cases are not data classes. Running our algorithms on this modified model using our multiple-view approach, the algorithms didn't identify these as anti-patterns. However, when we run our algorithms using a single view of use case diagram, then the algorithms falsely detect them as instance of creeping featurism anti-pattern.

8.6 Discussion

Table 35: Average of the algorithm results

	HC	LAHC	SA
Avg	239.81	239.77	239.75
StD	0.02	0.09	0.08

We run the three algorithms 15 times and compare their results. We can see from Table 35 that SA has the better average, followed by LAHC and followed by HC. It shows that SA in average performed better than the other algorithms by having a lower fitness function.

We run Wilcoxon pair signed test on each pair of these algorithms. Between HC and SA, we found that the p-value is $0.003 < 0.05$ which indicates that the results is significant at 95% level. Using the same test between LAHC and SA produced a p-value of $0.0232 < 0.05$ which signifies the result at 95% level too.

HC algorithm performed badly because of its starting position. Since the algorithm terminate quickly if no improvement to the objective function happens, it is necessary to give the algorithm some hints on how its starting position. In multiple-view diagram, the diagram consists of three diagrams and the anti-pattern is exhibited at the use case portion of the diagram. This poses a challenge for a basic algorithm such as an HC.

LAHC suffers from the initial random point as HC and terminates quickly. However, as we can see from runs 3 and 4, if the initial point is good, then it can find many others. In run 3, it was able to find 2 and in run 4, it was able to find 6 smells. The same improvement suggestion that was provided to HC in the previous section can be applied here. Moreover, the design of the memory list parameter might help in detecting many smells in the same run and prevents the algorithm from a quick termination.

SA performed very well in our multiple-view model. The design of an efficient cooling schedule determines the tolerance of the algorithm for non-improvement iterations. This can let the algorithm runs for extra few iterations before termination.

8.7 Threats of Validity

There are some threats that might affect the validity of the results mentioned above. These threats can be attributed to the: algorithm and to the multiple-view diagram. The algorithms are random in nature and thus the algorithm might start from different position in each run. In addition, the parameters of the algorithms produce different results if they are set with different values. There are no optimal values in the literature that drive the best performance of the algorithms in software refactoring. We have used a trial and error method to set the parameters. Moreover, the results returned by these algorithms are evaluated using some of the quality metrics of usecase, sequence and class diagrams. There is some controversy about whether these metrics capture the design quality or not. However, these metrics are used by different researchers including Misbhauddin.

Creeping Featurism is a use case diagram smell. Misbhauddin provided a mechanism to detect this smell in a multiple-view diagram. Our results are based on this mechanism.

In addition, we only investigate one smell in a small dataset. To extend the generality of this research, a large dataset with more than one smell can be used to ensure that the results here are generable. However, not so many smell in a multiple-view context available in the literature. In addition, it is not feasible always to find a large dataset showing the three diagrams of a system.

8.8 Conclusion

The above experiments show the applicability of three search-based algorithms in detecting and refactoring a design smell automatically in a multiple-view diagram. Multiple-view diagram poses a difficulty in the sense that the algorithm has to search for three diagrams in order to detect one smell. This difficulty is clearer for any greedy algorithm such as: HC due to the difficulty to continue of detecting further smells upon detecting the first one. HC terminates immediately if it fails to detect any smell in any of its iterations. We observed that LAHC, a variant of HC, is able to assist in that issue by its ability to continue finding other smells and prevent the algorithm from a quick termination. This attributes to its relaxation method. SA performed the best among these algorithms. This might be contributed to the cooling schedule parameters which allow the algorithm to continue running some iterations of search before the termination condition is met.

CHAPTER 9

CONTRIBUTIONS AND LIMIATIONS

9.1 Contribution

The previous sections laid out all the issues in AI-based software refactoring: the selection of techniques, the suitable metrics, the model transformation approaches and the application on the multi-view UML models. Leveraging the power of AI for refactoring is promising. Studying the applicability of various AI techniques over a set of different UML diagrams will surely enrich the domain. Testing our approach on different views of UML including the multi-view model will add another dimension to the Model-driven refactoring literature. In Addition, our techniques are extendable and scalable by implementing on other UML diagrams or models, and by improving the applied AI algorithms via operator and parameter tuning.

In summary, the major contributions of this research to the model-based literature are outlined below:

Contribution 1: Refactoring UML Diagrams: As inferred from literature, there are a few attempts to perform refactoring at the design level (UML diagrams) and the interest among research community is increasing recently. Some diagrams were more refactored than others and some diagrams were not targeted [8]. In addition, not all design smells or refactoring operations are studied. Since UML diagrams are designed to contribute to three views of development, we

opt to select one candidate from each view with a different design smell and refactoring operation.

Contribution 2: Refactoring the multi-view UML model: A multi-view UML model is a novel model incorporating diagrams from different views. As mentioned earlier, we selected Misbhauddin's multi-view model which unify class diagram, use case diagram and sequence diagram. We will detect any ill-structured designs or anti-patterns in the multi-view UML model and propose refactoring operations to enhance its quality.

***Contribution 3: Applying AI algorithms for refactoring UML models*:** Few studies applied different various automating techniques on refactoring. However, most of these studies are dedicated to code refactoring. There is much space where AI techniques might be useful. We are going to apply various AI algorithms to refactor the selected UML diagrams based on some quality metrics. A comparison between the performances of these algorithms is presented using some statistical techniques such as T-test and ANOVA. In addition, we will extend this approach to be applied to multi-view model.

Contribution 4: Refactoring UML diagrams using computing metrics: Most of the available metrics are suitable to code refactoring. Few of these metrics can be imported and used at design level. Additionally, some of the proposed metrics in the literature was not applied in refactoring such as [144]. We aim to perform refactoring based on competing metrics. For example, our refactoring operation will be applied in order to optimize coupling and cohesion simultaneously. We may also propose new metrics to measure the multi-view model.

Contribution 5: Comparison between refactoring UML models and multi-view UML model: Since the multi-view UML model is novel, there are no studies discussing its metrics. Hence, it

is necessary to figure out how to evaluate and compare the results obtained by refactoring individual UML models and the multi-view UML model. Since different algorithms are applied to different UML diagram, our focus will be shifted not only to compare the algorithms' performance but to evaluate the quality gain obtained by applying AI algorithms.

9.2 Limitations

At this stage, we are not able to anticipate all types and instances of limitations in our research. However, the following limitations emerged from our works so far. Knowing these limitations at an early stage, we tried to mitigate their effects on the validity of our research:

- Data is an issue in the field of software engineering. Many authors rely on a generated UML from an available source code [13, 14], others rely on published data [68]. Al-Dallal [78] reported the issue of the absence of repository for model refactoring. To mitigate this effect, we have collected data from three different sources as explained in "Data Collection" section. These sources are: open source, senior projects of KFUPM students and real-world case studies (either free or commercial).

- Metrics is one of our research tools to validate the results. They are some controversy on how these metrics reflect what they measure precisely and to which degree they are valid. To mitigate the effect of this limitation, we rely on the wide adoption of these metrics by many authors in the domain.

- Some of the Artificial Intelligence techniques do not show explicitly the steps on how the software is refactored. To mitigate that, an analysis with a brief description of the algorithm and its running steps is provided.

- In running our experiments, we are relying on our implementations of the algorithms. Even though, some templates are available in the web, adjustments are required. Thus, the implementations may be error-prone, and to mitigate that, we are going to run extensive testing to ensure the correctness of their implementation.

- We did not cover all constructs and notations of a UML diagram such as Abstraction and collaboration. We only focused on common UML constructs of use case , sequence and class diagrams.

- We used direct implementation of our algorithms on the UML data. This can pose some limitations for researchers who are interested in UML models at the metadata level.

- Our results on integrating different diagrams into a multiple-view diagram are based on one integration method proposed in the literature. There are various ways and methods that lead to an integrated diagram; however, we did not explore all of them.

APPENDIX A: Algorithm Templates

a. GA Template

Input: encoding of individuals

Process:

Initialize the population randomly;

Loop

Evaluate the objective function of each individual.

Loop:

Apply crossover & mutation operators.

Generate new children

Evaluate new children

End Loop if enough children are generated:

End Loop if termination criteria are satisfied

Output: optimum individual

b. Tabu Search Template

Input: encoding of the solution

Process:

Initialize the first solution randomly

Create an empty Tabu list T.

Loop:

Generate neighbors list N(x).

Select the best $n \in N$ and $n \notin T$.

$S \rightarrow n$

Update T

End Loop if termination criteria are satisfied:

Output: s

c. Ant Colony Optimization Template

Input: encoding the solution

Number of ants A

Process:

Initialize pheromone value

Loop:

Construct S from A

Update pheromone

Daemon Action

End loop if termination criteria are satisfied

Output: Best s ∈ S

d. Hill Climbing Template

Input: encoded the solution

Current node.

Process:

Set current node as the Max

Loop

Move to neighborhood

If neighborhood node > current node

Neighborhood node is the Max

Else

Move to another node

End loop if all nodes are examined

Output: Max

APPENDIX B: Smells and Anti-patterns

Model Smells [112]:

Table 36: Model Smells

Bad Smell	Definition
Lazy Class	Class is not doing enough or has less responsibility and has few functionality
Feature Envy	Class seems more interested in another class too much. That affects the design's flexibility
Middle Man	Class delegates between two or more classes
Message Chains	Class has long sequences of method calls or temporary variables to get routine data.
Long Method	Method has many statements. So, that method is difficult to read, understand and maintain
Long Parameter Lists	Method passes many parameters, the code more complex.
Switch Statement	Method indicates the duplicate source code and reveals a lack of object orientation.

Anti-patterns [93]:

Table 37: Anti-Pattern

Bad Smell	Definition
Blob (God Swiss)	It occurs for a class that has several data fields and functions, which result in complexity.
Lava Flow	Classes that is not coupled with other classes and has no interaction with other classes.
Functional Decomposition	A class that has only one function and it is private.
Poltergeists	It is a class that has a single method that has redundant navigation paths to all other classes.

APPENDIX C: Sequence Diagram (Similarity Matrix) classes).

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
M0	0	2	0	0	0	0	0	0	0	0
M1	1	0	2	0	0	0	0	0	0	0
M2	2	3	0	0	0	0	0	0	0	0
M3	1	0	0	0	0	0	0	0	0	0
M4	2	1	0	0	0	0	0	0	0	0
M5	1	0	0	0	0	0	0	0	0	0
M6	1	0	0	0	0	0	0	0	0	0
M7	2	0	0	0	0	0	0	0	0	0
M8	1	0	0	0	0	0	0	0	0	0
M9	1	0	0	0	0	0	0	0	0	0
M10	2	0	1	1	0	0	0	0	0	0
M11	0	2	0	0	0	0	0	0	0	0
M12	0	1	0	0	0	0	0	0	0	0
M13	0	2	0	0	0	0	0	0	0	0
M14	0	1	0	0	0	0	0	0	0	0
M15	0	0	1	0	0	0	0	1	0	0
M16	1	2	0	0	0	0	0	0	0	0
M17	0	1	0	0	0	0	0	0	0	0
M18	0	2	0	0	0	0	0	0	0	0
M19	0	1	0	0	0	0	0	0	0	0
M20	0	2	0	0	0	0	0	0	0	0

M21	0	2	0	0	0	0	0	0	0	0
M22	0	2	0	0	0	0	0	0	0	0
M23	0	1	0	0	0	0	0	0	0	0
M24	1	2	0	0	0	0	0	0	0	0
M25	1	2	0	0	0	0	0	0	0	0
M26	0	1	0	0	0	0	0	0	0	0
M27	1	0	0	0	0	0	0	0	0	0
M28	0	0	0	0	0	0	0	0	2	0
M29	0	1	0	0	0	0	0	0	0	0
M30	0	1	0	0	0	0	0	0	0	0
M31	0	1	0	0	0	0	0	0	0	0
M32	1	2	1	1	1	0	0	0	0	0
M33	0	2	1	0	0	0	0	0	0	0
M34	0	1	0	0	0	0	0	0	0	0
M35	0	3	0	0	0	0	0	0	0	0
M36	0	2	0	0	0	0	0	1	0	0
M37	0	1	0	0	0	0	0	0	0	0
M38	0	1	0	0	0	0	0	0	0	0
M39	0	1	0	0	0	0	0	0	0	0
M40	0	2	0	0	0	0	0	0	0	0
M41	0	0	1	0	0	0	0	0	0	0
M42	0	0	1	0	0	0	0	0	0	0
M43	0	0	2	0	0	0	0	0	0	0
M44	0	0	1	0	0	0	0	0	0	0

M45	0	0	3	0	0	0	0	1	0	0
M46	1	0	3	0	0	0	0	0	0	0
M47	0	1	2	0	0	0	0	0	0	0
M48	0	0	1	0	0	0	0	0	0	0
M49	0	0	1	0	0	0	0	0	0	0
M50	0	0	2	0	0	0	0	0	0	0
M51	0	0	1	0	0	3	0	0	0	0
M52	0	0	2	0	0	0	0	0	0	0
M53	0	0	2	0	0	0	0	0	0	0
M54	0	1	2	0	0	0	0	0	0	0
M55	0	0	1	0	0	0	0	0	2	0
M56	0	0	1	0	0	0	0	0	0	0
M57	0	0	2	0	0	0	0	0	0	0
M58	0	0	2	0	0	0	1	0	0	0
M59	0	0	2	0	1	0	0	0	0	0
M60	0	0	1	2	0	0	0	0	0	0
M61	0	0	0	1	0	0	0	0	0	0
M62	0	0	0	1	0	0	0	0	0	0
M63	0	0	0	1	0	1	0	0	0	0
M64	0	0	0	1	0	0	0	0	0	0
M65	0	0	1	2	0	0	0	0	0	0
M66	0	0	0	1	0	0	0	0	0	0
M67	0	0	0	1	0	0	0	0	0	0
M68	0	0	0	2	0	0	1	0	0	0

M69	0	0	0	2	0	0	0	0	0	0
M70	0	0	0	2	0	0	0	0	0	0
M71	0	0	0	3	0	0	0	0	1	0
M72	0	0	0	3	0	1	0	0	0	0
M73	1	0	0	2	0	0	0	0	0	0
M74	0	0	0	1	0	0	0	0	0	0
M75	0	0	0	1	0	0	0	0	0	0
M76	0	0	0	1	0	0	0	0	0	0
M77	0	0	0	2	0	0	0	0	0	0
M78	0	0	0	2	0	0	0	0	0	0
M79	0	0	0	2	0	0	0	0	0	1
M80	0	0	0	0	0	0	2	0	0	0
M81	0	0	0	0	0	0	1	0	0	0
M82	0	0	0	0	0	0	0	1	0	0
M83	0	1	0	0	2	0	0	0	0	0
M84	0	2	0	0	1	0	0	0	0	0
M85	0	1	0	0	2	0	0	0	1	0
M86	0	0	0	0	1	0	0	0	0	0
M87	0	0	0	0	1	0	0	0	0	0
M88	0	0	0	0	1	0	0	0	0	0
M89	0	0	0	0	1	0	0	0	0	0
M90	0	0	0	0	1	0	0	0	0	0
M91	0	0	0	0	1	0	0	2	0	0
M92	0	1	0	0	1	0	0	0	0	2

M93	0	2	0	0	1	0	0	0	0	0
M94	0	1	0	0	2	0	0	0	0	0
M95	0	0	0	0	3	0	0	1	0	0
M96	0	0	0	0	2	0	3	0	0	0
M97	0	0	0	0	2	0	0	0	0	0
M98	0	0	0	0	2	0	0	0	0	0
M99	0	0	0	0	1	0	0	0	0	0

APPENDIX D: Eclipse Classes Metric Values

Class Name	V(G)	DIT	NOA	NLM	WMC	RFC	DAC	CBO	LCOM	LOC
AbstractMethodDeclaration	11	2	17	28	99	32	1	5	216	364
AbstractVariableDeclaration	5	2	16	11	29	14	0	1	53	78
AddFromHistoryAction	8	2	1	3	24	7	1	5	6	104
Argument	10	2	1	8	48	11	0	3	28	140
ArrayAllocationExpression	18	2	3	5	48	8	1	5	0	126
ArrayInitializer	18	2	2	5	60	7	0	3	0	178
BinaryCompareViewer	12	2	7	4	27	6	2	4	2	105
BooleanConstant	1	3	3	5	926	6	0	1	4	24
BufferedContent	3	2	2	8	24	8	0	1	4	53
ByteConstant	1	3	1	11	926	12	0	1	0	40
Canvas	9	2	11	17	76	30	2	2	0	207
Caret	6	2	1	3	19	3	0	0	0	258
ChangePropertyAction	3	2	4	8	24	8	2	2	6	57
ClassFileReader	46	3	25	45	213	76	3	21	600	801

ClassFileStruct	5	2	3	9	24	12	0	3	0	71
ClassFormatException	3	4	34	6	28	10	1	4	3	94
ClasspathDirectory	10	2	4	10	46	27	2	6	24	157
CodeFormatter	1	2	9	3	12	3	0	1	3	20
Color	29	2	2	5	37	9	0	3	0	78
ColorDialog	1	2	106	0	11	0	1	1	0	58
CompareAction	3	2	3	3	17	3	0	0	0	34
CompareConfiguration	4	2	21	33	54	52	4	8	404	341
CompareMessages	7	2	5	7	28	11	0	2	5	115
CompareUI	2	2	9	24	39	25	1	4	276	113
CompareViewerPane	3	2	5	20	41	20	0	2	168	170
CompareViewerSwitchingPane	11	3	4	14	83	14	1	2	23	200
CompilerOptions	181	2	236	15	374	23	1	7	15	1332
ConditionalFlowInfo	2	3	2	36	66	36	0	1	0	160
Constant	5	2	1	34	925	37	0	3	219	1365
Context	7	2	1	5	33	17	1	2	8	121
Cursor	20	2	35	20	113	36	4	10	148	395
DebugEvent	1	2	7	0	11	0	0	0	0	129

DebugException	9	2	7	37	107	49	3	10	308	15
Device	14	2	4	6	46	13	0	4	0	405
DiffContainer	2	2	2	6	17	6	0	1	11	49
DiffElement	9	4	6	19	75	22	0	2	92	31
DiffNode	7	2	8	26	57	33	2	4	161	179
DirectoryDialog	14	2	2	5	30	7	1	1	0	109
DocLineComparator	6	2	3	4	26	9	1	3	0	110
DocumentManager	3	2	1	8	24	12	0	5	26	42
DocumentRangeNode	3	3	21	7	38	10	0	2	5	213
ExceptionHandler	7	2	6	7	26	9	1	4	5	62
ExceptionHandlingFlow Context	8	3	10	9	177	15	1	5	4	204
FieldInfo	8	3	9	23	99	45	1	14	111	314
FileDialog	40	2	8	14	86	28	1	2	76	329
FileFinder	6	2	0	2	18	10	0	4	1	28
FlowInfo	7	2	8	44	32	44	0	1	936	172
Font	11	2	7	11	38	28	1	5	4	208
FontData	3	2	5	8	19	8	0	0	0	164
FontMetrics	5	2	1	24	75	25	0	2	0	41
Group	3	2	1	5	27	12	1	1	39	85
HistoryItem	3	2	2	7	20	7	0	2	3	47

HrefUtil	1	2	2	2	13	2	1	2	0	70
ImageMergeViewer	10	2	9	6	30	6	2	3	7	100
InitializationFlowContext	2	4	5	3	179	7	1	5	0	74
InnerClassInfo	4	3	10	5	39	9	0	2	0	82
IntConstant	1	3	17	11	942	12	0	2	0	74
JDTCompilerAdapter	34	2	10	6	87	37	3	22	0	392
Label	9	2	3	7	63	18	1	1	92	225
LabelFlowContext	3	4	1	2	149	4	0	1	0	30
Launch	8	2	5	10	38	18	1	4	35	333
Link	7	2	11	18	40	22	2	3	137	21
LongConstant	3	3	3	11	928	12	0	2	0	47
LoopingFlowContext	41	4	16	11	295	22	1	5	34	550
MessageBox	23	2	1	3	60	7	1	1	4	160
MethodInfo	17	3	12	26	121	35	0	4	249	404
NavigationAction	6	2	2	2	18	3	0	2	0	47
OverlayIcon	5	2	6	14	33	16	1	4	43	95
PDERuntimePlugin	2	2	58	5	17	6	1	3	4	108
PDERuntimePluginImages	22	2	25	5	82	8	0	4	6	97
ProgressBar	4	2	0	9	31	12	0	0	66	79

RecoveredBlock	22	4	9	20	143	25	1	11	35	324
RecoveredField	18	3	8	12	123	16	1	7	0	234
RecoveredImport	2	3	1	6	77	6	0	1	0	31
RecoveredInitializer	9	4	7	13	134	17	2	8	6	249
RecoveredMethod	23	3	13	20	178	27	2	9	0	514
Region	12	2	2	7	78	23	1	2	0	177
RegistryBrowserLabelProvider	4	2	1	9	29	11	0	2	0	325
ResizableDialog	3	2	2	4	18	4	0	1	2	122
ResourceNode	5	3	2	16	52	19	1	3	0	119
Sash	13	2	9	3	80	20	2	4	128	295
Scale	4	2	2	13	47	18	0	1	132	138
Scrollable	8	2	3	5	89	28	0	0	62	278
SimpleTextViewer	4	3	2	6	31	7	1	2	0	42
StringConstant	1	3	1	4	926	4	1	1	4	19
TabItem	11	2	4	9	48	16	1	1	33	152
Toc	2	2	6	7	19	7	1	3	15	154
TocFile	3	3	1	1	31	3	0	6	0	47
TocFileParser	6	2	9	16	59	35	2	16	20	36
TocManager	3	2	0	6	24	12	0	6	15	261
ToolBar	7	2	2	9	99	34	0	3	125	324

URLCoder	6	2	5	6	34	6	0	0	3	66

APPENDIX E: Android Classes Metric Values

Class Name	V(G)	DIT	NOA	NLM	WMC	RFC	DAC	CBO	LCOM	LOC
AbstractMethodError	1	6	1	0	21	0	0	1	0	10
AccelerateInterpolator	2	2	2	1	13	2	0	1	0	30
ActivityNotFoundException	1	5	0	0	21	0	0	1	0	11
AlarmManager	1	2	10	6	17	6	0	1	0	59
AlgorithmParameterGenerator	3	2	6	10	26	11	1	2	23	80
AppWidgetProvider	8	2	0	5	23	5	0	1	10	42
ArrayStoreException	1	5	1	0	21	0	0	1	0	10
AssertionError	1	4	1	0	21	7	0	0	0	30
AuthProvider	1	2	1	3	11	3	0	4	3	13
BaseInputConnection	18	2	9	29	126	29	1	2	174	458
CharArrayBuffer	1	2	2	0	11	0	0	0	0	11
ClassFormatError	1	5	1	0	21	0	0	1	0	10
CollationElementIterator	1	2	2	11	22	11	0	1	0	43
ContentUris	1	2	0	3	14	4	0	2	3	14

CursorIndexOutOfBounds Exception	1	6	0	0	21	0	0	1	0	9
DataSetObserver	1	2	0	2	13	2	0	0	1	7
DeleteEventHelper	6	2	18	4	27	4	1	2	0	190
DocumentBuilder	3	2	1	14	23	20	0	15	91	85
EmailAddressValidator	1	2	0	2	13	2	0	1	1	11
Environment	2	2	27	15	30	18	2	5	92	110
Error	1	3	1	0	21	0	0	1	0	16
Exception	1	3	1	0	21	0	0	1	0	16
FactoryConfigurationError	2	4	1	2	23	2	1	2	0	30
FocusFinderHelper	1	2	1	6	17	6	0	0	3	26
GpsSatellite	1	2	8	7	19	8	0	0	16	44
Gravity	15	2	23	5	52	5	0	0	10	159
HapticFeedbackConstants	1	2	7	0	11	0	0	0	0	11
IllegalAccessError	1	6	1	0	21	0	0	1	0	10
IllegalAccessException	1	4	1	0	21	0	0	1	0	10
IllegalStateException	1	5	1	0	21	0	0	1	0	16
InCallMenu	9	2	18	1	29	5	1	1	0	262

InflateException	1	5	0	0	21	0	0	1	0	15
LocationProvider	2	2	7	11	13	12	1	1	53	40
Math	12	2	3	58	118	71	1	5	1651	405
MessagingListener	1	2	0	19	30	19	0	2	171	56
MutableContextWrapper	1	2	0	1	12	1	0	0	0	9
NotificationManager	3	2	5	6	22	6	1	1	0	71
Number	1	2	1	6	13	6	0	0	15	16
OnScreenHint	5	2	14	6	26	6	2	3	0	111
OutOfMemoryError	1	5	1	0	21	0	0	1	0	10
ParserConfigurationException	1	4	0	0	21	0	0	1	0	9
Process	0	2	0	6	11	6	0	3	15	11
ReceiverCallNotAllowedException	1	2	0	0	11	0	0	1	0	7
Ringtone	10	2	11	8	42	12	2	5	0	159
RuntimeException	1	4	1	0	21	0	0	1	0	16
RuntimePermission	1	4	16	0	18	0	0	1	0	41
SAXParser	3	2	1	19	38	35	0	17	169	159

SecurityException	1	5	1	0	21	0	0	1	0	16
SensorEvent	1	2	4	0	11	0	0	0	0	10
SoundEffectConstants	1	2	5	1	18	1	0	0	0	25
SpecialCharSequenceMgr	4	2	3	7	33	15	1	3	39	124
SQLException	1	5	0	0	21	0	0	1	0	9
SQLiteAbortException	1	3	0	0	11	0	0	1	0	7
SQLiteClosable	4	2	2	6	22	8	0	3	9	51
SQLiteConstraintException	1	3	0	0	11	0	0	1	0	7
SQLiteDatabaseCorruptException	1	3	0	0	11	0	0	1	0	7
SQLiteDiskIOException	1	3	0	0	11	0	0	1	0	7
SQLiteDoneException	1	3	0	0	11	0	0	1	0	7
SQLiteException	1	2	0	0	11	0	0	1	0	8
SQLiteFullException	1	3	0	0	11	0	0	1	0	7
SQLiteMisuseException	1	3	0	0	11	0	0	1	0	7
SQLiteOpenHelper	12	2	7	6	33	7	1	4	9	114
SQLiteStatement	2	2	0	7	19	7	0	1	21	79
StackOverflowError	1	5	1	0	21	0	0	1	0	10
StaleDataException	1	5	0	0	21	0	0	1	0	12

StatusBarManager	1	2	8	6	17	6	0	1	0	64
StrictMath	8	2	3	58	86	90	1	6	1653	283
StringIndexOutOfBoundsException	1	6	1	0	21	0	0	1	0	13
SyncContext	3	2	3	4	18	4	0	1	0	38
SystemProperties	3	2	2	12	24	13	0	1	36	52
ThreadDeath	1	4	1	0	21	0	0	0	0	6
TouchDelegate	6	2	9	1	21	1	0	0	0	64
TypedArray	6	2	7	31	94	37	0	8	0	393
TypeNotPresentException	1	5	2	1	22	1	1	1	0	12
UnknownError	1	5	1	0	21	0	0	1	0	10
VerifyError	1	5	1	0	21	0	0	1	0	10
Vibrator	3	2	3	3	18	3	1	1	0	52
WindowManagerImpl	9	2	17	14	54	17	0	3	54	249

References

[1] OMG, "OMG Unified Modeling LanguageTM (OMG UML), Superstructure," ed.

[2] *http://www.agilemodeling.com/artifacts/classDiagram.htm*.

[3] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, 1 edition ed. Reading, MA: Addison-Wesley Professional, 1999.

[4] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering,* vol. 30, pp. 126-139, February 2004

[5] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology,* vol. 51, pp. 1319-1326, September 2009

[6] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 20, pp. 345-364, 2008.

[7] A. Alkhalid, *et al.*, "Software refactoring at the package level using clustering techniques," *IET Software,* vol. 5, pp. 276-284, 2011/06// 2011.

[8] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empirical Software Engineering,* accepted 2013 accepted 2013.

[9] E. Song, *et al.*, "Using roles for pattern-based model refactoring," in *Proceedings of the Workshop on Critical Systems Development with UML (CSDUML'02)*, 2002.

[10] T. Massoni, *et al.*, "Formal Refactoring for UML Class Diagrams," 2005, pp. 152-167.

[11] T. Mens, "On the Use of Graph Transformations for Model Refactoring," in *Generative and Transformational Techniques in Software Engineering*, R. Lämmel, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2006, pp. 219-257.

[12] G. Sunyé, *et al.*, "Refactoring UML Models," in *≪UML≫2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds., ed: Springer Berlin Heidelberg, 2001, pp. 134-148.

[13] A. Ghannem, *et al.*, "Model Refactoring Using Interactive Genetic Algorithm," in *Search Based Software Engineering*, G. Ruhe and Y. Zhang, Eds., ed: Springer Berlin Heidelberg, 2013, pp. 96-110.

[14] A. Ghannem, *et al.*, "Detecting Model Refactoring Opportunities Using Heuristic Search," 2011, pp. 175-187.

[15] A. A. Issa, "Utilising Refactoring To Restructure Use-Case Models," *Lecture Notes in Engineering and Computer Science*, 2007.

[16] M. Misbhauddin, "Toward An Integarted Metamodel Based Approach for Software Refactoring," 2012.

[17] J. Rumbaugh, *et al.*, *The Unified Modeling Language Reference Manual,*, 2 edition ed. Boston: Addison-Wesley Professional, 2004.

[18] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, 1 edition ed. New York: Wiley, 2003.

[19] Y. Singh and M. Sood, "Model Driven Architecture: A Perspective," in *Advance Computing Conference, 2009. IACC 2009. IEEE International*, 2009, pp. 1644-1652.

[20] *http://en.wikipedia.org/wiki/Sequence_diagram#mediaviewer/File:CheckEmail.svg*.

[21] J. Iivari, "Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis," *Information and Software Technology*, vol. 37, pp. 155-163, 1995.

[22] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empirical Software Engineering*, pp. 1-46, October 15, 2013 2013.

[23] E. Koc, *et al.*, "An Empirical Study About Search-Based Refactoring Using Alternative Multiple and Population-Based Search Techniques," in *Computer and Information Sciences II*, E. Gelenbe, *et al.*, Eds., ed: Springer London, 2012, pp. 59-66.

[24] A. Ouni, *et al.*, "The Use of Development History in Software Refactoring Using a Multi-objective Evolutionary Algorithm," 2013, pp. 1461-1468.

[25] Y. A. Khan and M. El-Attar, "Using model transformation to refactor use case models based on antipatterns," *Information Systems Frontiers*, pp. 1-34, 2014/08/13 2014.

[26] M. El-Sharqwi, *et al.*, "Pattern-based model refactoring," in *2010 International Conference on Computer Engineering and Systems (ICCES)*, 2010, pp. 301-306.

[27] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Softw.*, vol. 20, pp. 42–45, September 2003 2003.

[28] M. Kolp, *et al.*, "Multi-agent architectures as organizational structures," *Autonomous Agents and Multi-Agent Systems,* vol. 13, pp. 3-25, 2006.

[29] M. Misbhauddin and M. Alshayeb, "Model-Driven Refactoring Approaches: A Comparison Criteria," in *2012 African Conference on Sofware Engineering and Applied Computing (ACSEAC)*, 2012, pp. 34-39.

[30] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software & Systems Modeling,* vol. 9, pp. 141-160, 2010/04/01/ 2010.

[31] K. Czarnecki. (2003). *Classification of Model Transformation Approaches*.

[32] T. Massoni, *et al.*, "Formal Refactoring for UML Class Diagrams," 2005, pp. 152–167.

[33] C. Wohlin, *et al.*, *Experimentation in software engineering*: Springer, 2012.

[34] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: PWS Publishing Co., 1998.

[35] H. Kim and C. Boldyreff, "Developing Software Metrics Applicable to UML Models," 2002.

[36] J. A. McQuillan and J. F. Power, *Some observations on the application of software metrics to UML models,* 2006.

[37] J. A. Mcquillan and J. F. Power, "A definition of the Chidamber and Kemerer metrics suite for the Unified Modeling Language," 2006 2006.

[38] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.,* vol. 20, pp. 476–493, June 1994 1994.

[39] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering,* vol. 28, pp. 4-17, 2002/01// 2002.

[40] M. Gendreau and J.-Y. Potvin, *Handbook of metaheuristics* vol. 2: Springer, 2010.

[41] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*: MIT Press, 1992.

[42] H. Mühlenbein and G. Paaß, "From recombination of genes to the estimation of distributions I. Binary parameters," in *Parallel Problem Solving from Nature —*

PPSN IV. vol. 1141, H.-M. Voigt, *et al.*, Eds., ed: Springer Berlin Heidelberg, 1996, pp. 178-187.

[43] W.-D. Chang, "An improved real-coded genetic algorithm for parameters estimation of nonlinear systems," *Mechanical Systems and Signal Processing*, vol. 20, pp. 236-246, January 2006 2006.

[44] Y.-C. Chuang and C.-T. Chen, "A study on real-coded genetic algorithm for process optimization using ranking selection, direction-based crossover and dynamic mutation," in *2011 IEEE Congress on Evolutionary Computation (CEC)*, 2011, pp. 2488-2495.

[45] W. Hare, *et al.*, "A Survey of Non-gradient Optimization Methods in Structural Engineering," *Adv. Eng. Softw.*, vol. 59, pp. 19–28, May 2013 2013.

[46] Y. Xin-She and S. Deb, "Cuckoo Search via Lévy flights," in *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, 2009, pp. 210-214.

[47] M. Dorigo and T. Stützle, *Ant Colony Optimization*: MIT Press, 2004.

[48] Z. N. Azimi, "Comparison of metaheuristic algorithms for Examination Timetabling Problem," *Journal of Applied Mathematics and Computing*, vol. 16, pp. 337-354, 2004/03/01 2004.

[49] H. Bai and B. Zhao, "A Survey on Application of Swarm Intelligence Computation to Electric Power System," in *The Sixth World Congress on Intelligent Control and Automation, 2006. WCICA 2006*, 2006, pp. 7587-7591.

[50] F. Chicano, *et al.*, "Comparing Metaheuristic Algorithms for Error Detection in Java Programs," in *Search Based Software Engineering*, M. B. Cohen and M. Ó. Cinnéide, Eds., ed: Springer Berlin Heidelberg, 2011, pp. 82-96.

[51] C. García-Martínez, *et al.*, "A taxonomy and an empirical analysis of multiple objective ant colony optimization algorithms for the bi-criteria TSP," *European Journal of Operational Research*, vol. 180, pp. 116-148, July 1, 2007 2007.

[52] I. Chaparro and F. Valdez, "Variants of Ant Colony Optimization: A Metaheuristic for Solving the Traveling Salesman Problem," in *Recent Advances on Hybrid Intelligent Systems*. vol. 451, O. Castillo, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2013, pp. 323-331.

[53] D. Mukherjee and S. Acharyya, "New variants of Ant Colony Optimization for Network Routing," in *Computer and Information Technology (ICCIT), 2011 14th International Conference on*, 2011, pp. 445-450.

[54] X.-S. Yang, "A New Metaheuristic Bat-Inspired Algorithm," in *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*. vol. 284, J. González, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 65-74.

[55] X.-S. Yang, "Firefly Algorithm, Lévy Flights and Global Optimization," in *Research and Development in Intelligent Systems XXVI*, M. Bramer, *et al.*, Eds., ed: Springer London, 2010, pp. 209-218.

[56] D. Karaboga, "An idea based on honey bee swarm for numerical optimization," Technical report-tr06, Erciyes university, engineering faculty, computer engineering department2005.

[57] F. Glover, "Future Paths for Integer Programming and Links to Artificial Intelligence," *Comput. Oper. Res.*, vol. 13, pp. 533–549, May 1986 1986.

[58] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*: Prentice Hall, 2010.

[59] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, pp. 338-353, 1965.

[60] C.-T. Lin, *et al.*, "Fuzzy neural network design using support vector regression for function approximation with outliers," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, 2005, pp. 2763 - 2768 Vol. 3.

[61] L. A. Zadeh and J. Kacprzyk, *Fuzzy logic for the management of uncertainty*: Wiley, 1992.

[62] L.-X. Wang, "Universal approximation by hierarchical fuzzy systems," *Fuzzy Sets and Systems*, vol. 93, pp. 223-230, January 16, 1998 1998.

[63] J. Han, *et al.*, *Data Mining: Concepts and Techniques, Second Edition (The Morgan Kaufmann Series in Data Management Systems)*: Morgan Kaufmann, 2006.

[64] W. Stadler, *Multicriteria Optimization in Engineering and in the Sciences* vol. 37: Springer, 1988.

[65] M. Affenzeller, *et al.*, "Metaheuristic Optimization," in *Hagenberg Research*, B. Buchberger, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2009, pp. 103-155.

[66] E.-G. Talbi, *Metaheuristics: from design to implementation*. Hoboken, N.J.: John Wiley & Sons, 2009.

[67] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," 1992 1992.

[68] E. Song, *et al.*, "Using Roles for Pattern-Based Model Refactoring," 2002.

[69] T. Mens and A. V. Deursen, *Refactoring: Emerging Trends and Open Problems*, 2003.

[70] B. D. Bois, *et al.*, "A Discussion of Refactoring in Research and Practice," 2004 2004.

[71] F. Simon, *et al.*, "Metrics based refactoring," in *Fifth European Conference on Software Maintenance and Reengineering, 2001.*, 2001, pp. 30-38.

[72] R. Reißing, "Towards a model for object-oriented design measurement," in *5th International ECOOP workshop on quantitative approaches in object-oriented software engineering*, 2001, pp. 71-84.

[73] N. Fenton, "Software measurement: a necessary scientific basis," *IEEE Transactions on Software Engineering*, vol. 20, pp. 199-206, 1994/03// 1994.

[74] R. Lincke and W. Löwe, *Foundations for Defining Software Metrics*, 2006.

[75] E. Domínguez, *et al.*, "A Survey of UML Models to XML Schemas Transformations," in *Web Information Systems Engineering – WISE 2007*, B. Benatallah, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2007, pp. 184-195.

[76] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125-142, March 27, 2006 2006.

[77] M. Misbhauddin and M. Alshayeb, "Model-Driven Refactoring Approaches: A Comparison Criteria," in *Sofware Engineering and Applied Computing (ACSEAC), 2012 African Conference on*, 2012, pp. 34-39.

[78] J. AlDallal, "Identifiying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review," *Information and Software Technology*, 2014 2014.

[79] T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1 edition ed. New York: Wiley, 1998.

[80] M. Fowler. (2013, 21-Oct-2014). *Catalog of Refactorings*. Available: http://www.refactoring.com/catalog/

[81] Y. Kataoka, *et al.*, "Automated Support for Program Refactoring using Invariants," presented at the Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), 2001.

[82] S. W. Fowler, *et al.*, "Virtually Embedded Ties," *Journal of Management*, vol. 30, pp. 647-666, October 2004 2004.

[83] A. Issa, "Utilising Refactoring: To Restructure Use-Case Models," in *Proceedings of the World Congress on Engineering (WCE)*, London, U.K, 2007.

[84] K. Phalp, *et al.*, "Assessing the quality of use case descriptions," *Software Quality Journal*, vol. 15, pp. 69-97, 2007/03/01 2007.

[85] W. Yu, *et al.*, "Refactoring Use Case Models on Episodes," in *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, 2004, pp. 328-331.

[86] L. Xu and G. Butler, "Cascaded refactoring for framework development and evolution," in *Proceedigns of the Australian Software Engineering Conference* 2006, p. 10 pp.

[87] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software and Systems Modeling*, vol. 9, pp. 141-160, 2010.

[88] M. El-Attar and J. Miller, "Constructing high quality use case models: a systematic review of current practices," *Requirements Engineering*, vol. 17, pp. 187-201, 2012/09/01 2012.

[89] T. Mens, *et al.*, "Refactoring: Current Research and Future Trends," *Electronic Notes in Theoretical Computer Science*, vol. 82, pp. 483-499, December 2003 2003.

[90] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empirical Software Engineering*, vol. 20, pp. Page 206-251, 2013.

[91] J. Al Dallal, "Identifiying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review," *Information and Software Technology*, 2014 2014.

[92] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *Eighth International Joint Conference on Computer Science and Software Engineering*, 2011, pp. 331-336.

[93] R. Fourati, *et al.*, "A Metric-Based Approach for Anti-pattern Detection in UML Designs," in *Computer and Information Science 2011*, R. Lee, Ed., ed: Springer Berlin Heidelberg, 2011, pp. 17-33.

[94] A. Alkhalid, *et al.*, "Software refactoring at the class level using clustering techniques," *Journal of Research and Practice in Information Technology,* vol. 43, pp. 285-306, 2011.

[95] A. Alkhalid, *et al.*, "Software Refactoring at the Function Level Using New Adaptive K-Nearest Neighbor Algorithm," *Advances in Engineering Software,* vol. 41, pp. 1160-1178, 2010.

[96] D. Sahin, *et al.*, "Code-Smell Detection as a Bilevel Problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, pp. 1-44, 2014.

[97] A. Ouni, *et al.*, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering,* vol. 20, pp. 47-79, 2013/03/01 2013.

[98] M. W. Mkaouer and M. Kessentini, "Chapter Four - Model Transformation Using Multiobjective Optimization," in *Advances in Computers*. vol. Volume 92, H. Ali, Ed., ed: Elsevier, 2014, pp. 161-202.

[99] A. Ouni, *et al.*, "Chapter Four - Multiobjective Optimization for Software Refactoring and Evolution," in *Advances in Computers*. vol. Volume 94, H. Ali, Ed., ed: Elsevier, 2014, pp. 103-167.

[100] M. Kessentini, *et al.*, "Design Defects Detection and Correction by Example," in *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*, 2011, pp. 81-90.

[101] B. Amal, *et al.*, "On the Use of Machine Learning and Search-Based Software Engineering for Ill-Defined Fitness Function: A Case Study on Software Refactoring," in *Search-Based Software Engineering*. vol. 8636, C. Le Goues and S. Yoo, Eds., ed: Springer International Publishing, 2014, pp. 31-45.

[102] Y. Helio Yang, "Software quality management and ISO 9000 implementation," *Industrial Management & Data Systems,* vol. 101, pp. 329-338, 2001.

[103] S. N. Bhatti, "Why quality?: ISO 9126 software quality metrics (Functionality) support by UML suite," *ACM SIGSOFT Software Engineering Notes,* vol. 30, pp. 1-5, 2005.

[104] B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target," *IEEE Software,* vol. 13, p. 12, 1996.

[105] L. Erlikh, "Leveraging Legacy System Dollars for E-Business," *IT Professional,* vol. 2, pp. 17-23, 2000.

[106] J. R. Horgan, *et al.*, "Achieving software quality with testing coverage measures," *Computer,* vol. 27, pp. 60-69, 1994.

[107] E. Erturk and E. A. Sezer, "A comparison of some soft computing methods for software fault prediction," *Expert Systems with Applications,* vol. 42, pp. 1872-1879, 2015.

[108] E. Giger, *et al.*, "Method-level bug prediction," presented at the Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, Lund, Sweden, 2012.

[109] S. Kanmani, *et al.*, "Object-oriented software fault prediction using neural networks," *Information and Software Technology,* vol. 49, pp. 483-492, 2007.

[110] I. H. Laradji, *et al.*, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology,* vol. 58, pp. 388-402, 2015.

[111] H. A. Al-Jamimi and L. Ghouti, "Efficient prediction of software fault proneness modules using support vector machines and probabilistic neural networks," in *Software Engineering (MySEC), 2011 5th Malaysian Conference in*, 2011, pp. 251-256.

[112] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2011, pp. 331-336.

[113] S. S. Rathore and S. Kumar, "Predicting Number of Faults in Software System using Genetic Programming," *Procedia Computer Science,* vol. 62, pp. 303-311, 2015.

[114] A. ben Fadhel, *et al.*, "Search-based detection of high-level model changes," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 2012, pp. 212-221.

[115] A. Ghannem, *et al.*, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal,* pp. 1-19, 2015/03/10 2015.

[116] R. Mahouachi, *et al.*, "A new design defects classification: marrying detection and correction," presented at the Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering, Tallinn, Estonia, 2012.

[117] N. Moha, "Detection and correction of design defects in object-oriented designs," presented at the Companion to the 22nd ACM SIGPLAN conference on Object-

oriented programming systems and applications companion, Montreal, Quebec, Canada, 2007.

[118] A. M. AL-Kandari and K. M. EL-Naggar, "A genetic-based algorithm for optimal estimation of input–output curve parameters of thermal power plants," *Electrical Engineering*, vol. 89, pp. 585-590, 2007/09/01 2007.

[119] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, pp. 1120-1128, 2007.

[120] R. Malhotra, *et al.*, "On the applicability of evolutionary computation for software defect prediction," in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, 2014, pp. 2249-2257.

[121] R. Mahouachi, *et al.*, "Search-Based Refactoring Detection Using Software Metrics Variation," in *Search Based Software Engineering*. vol. 8084, G. Ruhe and Y. Zhang, Eds., ed: Springer Berlin Heidelberg, 2013, pp. 126-140.

[122] M. D'Ambros, *et al.*, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th International Conference on*, 2010, pp. 23-31.

[123] K. Gao, *et al.*, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, pp. 579-606, 2011.

[124] I. Gondra, "Applying machine learning to software fault-proneness prediction," *Journal of Systems and Software*, vol. 81, pp. 186-195, 2008.

[125] A. Maiga, *et al.*, "Support vector machines for anti-pattern detection," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, 2012, pp. 278-281.

[126] F. Arcelli Fontana, *et al.*, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, pp. 1-49, 2015.

[127] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, pp. 649-660, 2008.

[128] T. M. Khoshgoftaar, *et al.*, "Attribute selection and imbalanced data: Problems in software defect prediction," in *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, 2010, pp. 137-144.

[129] B. Ghotra, *et al.*, "Revisiting the impact of classification techniques on the performance of defect prediction models," presented at the Proceedings of the 37th International Conference on Software Engineering - Volume 1, Florence, Italy, 2015.

[130] P. K. Dhillon and G. Sidhu, "Can Software Faults be Analyzed using Bad Code Smells?: An Empirical Study," *International Journal of Scientific and Research Publications,* vol. 2, pp. 1-7, 2012.

[131] R. Ferenc, "Bug Forecast: A Method for Automatic Bug Prediction," in *Advances in Software Engineering*, ed: Springer, 2010, pp. 283-295.

[132] T. Hall, *et al.*, "Some Code Smells Have a Significant but Small Effect on Faults," *ACM Trans. Softw. Eng. Methodol.,* vol. 23, pp. 1-39, 2014.

[133] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: an empirical study," presented at the Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 2013.

[134] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software,* vol. 86, pp. 2639-2653, 2013.

[135] A. Sturm, "Guiding System Modelers in Multi View Environments: A Domain Engineering Approach," in *Proceedings of EMMSAD*, 2008, p. 131.

[136] R. France and J. Bieman, "Multi-view software evolution: a UML-based framework for evolving object-oriented software," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, 2001, p. 386.

[137] C. Gomez, *et al.*, "Multi-view Power Modeling Based on UML, MARTE and SysML," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, 2012, pp. 17-20.

[138] R. E. Lopez-Herrejon and A. Egyed, "Detecting inconsistencies in multi-view models with variability," in *Modelling Foundations and Applications*, ed: Springer, 2010, pp. 217-232.

[139] A. E. Khadija El Miloudi, "A Multi-View Approach for Formalizing UML State Machine Diagrams Using Z Notation," *WSEAS Transaction on Computers,* vol. 14, pp. 72-78, 2015.

[140] A. Alkhalid, *et al.*, "Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm," *Advances in Engineering Software,* vol. 41, pp. 1160-1178, 2010/10// 2010.

[141] Z. Marian, *et al.*, "Software Package Refactoring using a hierarchical clustering-based approach," *Journal of Systems and Software*, 2014 2014.

[142] I. H. Moghadam and M. Ó Cinnéide, "Code-Imp: A Tool for Automated Search-based Refactoring," 2011, pp. 41-44.

[143] D. Fonte, *et al.*, "Modeling Languages: metrics and assessing tools," 2012.

[144] M. Marchesi, "OOA metrics for the Unified Modeling Language," in *Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on*, 1998, pp. 67-73.

[145] R. Seidl and H. Sneed. (2010) Modeling Metrics for UML Diagrams. *Testing Experience*.

[146] T. McCabe and C. Butler, "Design complexity measurement and testing," *Communications of the ACM*, vol. 32, pp. 1415-1425, 1989.

[147] E. K. Burke and Y. Bykov, "A Late Acceptance Strategy in Hill-Climbing for Exam Timetabling Problems," in *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling*, Montreal, Canada, 2008.

[148] S. Kirkpatrick, *et al.*, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, May 13, 1983 1983.

[149] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 476-493, 1994.

[150] J. Al Dallal, "Measuring the discriminative power of object-oriented class cohesion metrics," *IEEE Transactions on Software Engineering*, vol. 37, pp. 788-804, 2011.

[151] C.-H. Lung, *et al.*, "Program restructuring using clustering techniques," *Journal of Systems and Software*, vol. 79, pp. 1261-1279, 2006.

[152] E. G. Talbi, "A Taxonomy of Hybrid Metaheuristics," *Journal of Heuristics*, vol. 8, pp. 541-564, 2002/09/01 2002.

[153] L. Jourdan, *et al.*, "Using Datamining Techniques to Help Metaheuristics: A Short Survey," in *Hybrid Metaheuristics*, F. Almeida, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2006, pp. 57-69.

[154] G. Perim, *et al.*, "K-Means Initialization Methods for Improving Clustering by Simulated Annealing," in *Advances in Artificial Intelligence – IBERAMIA 2008*. vol. 5290, H. Geffner, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 133-142.

[155] J. Liu and T. Liu, "Detecting community structure in complex networks using simulated annealing with -means algorithms," *Physica A: Statistical Mechanics and its Applications,* vol. 389, pp. 2300-2309, 2010.

[156] J. Krall, *et al.*, "GALE: Geometric Active Learning for Search-Based Software Engineering," *IEEE Transactions on Software Engineering,* vol. 41, p. 1001, 2015.

[157] M. Barros and A. Dias-Neto, "Threats to validity in search-based software engineering empirical studies," *Relatórios Técnicos do DIA/UNIRIO,* 2011.

[158] M. R. Hossain, *et al.*, "The Effectiveness of Feature Selection Method in Solar Power Prediction," *Journal of Renewable Energy,* vol. 2013, p. 9, 2013.

[159] M. Alshayeb, *et al.*, "Object-Oriented Class Stability Prediction: A Comparison Between Artificial Neural Network and Support Vector Machine," *Arabian Journal for Science and Engineering,* vol. 39, pp. 7865-7876, 2014.

[160] H. K. Wright, *et al.*, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP workshop on Future of software engineering research,* 2010, pp. 411-414.

[161] E. K. Burke and Y. Bykov, "A late acceptance strategy in hill-climbing for exam timetabling problems," in *PATAT 2008 Conference, Montreal, Canada,* 2008.

[162] E. Koc, *et al.*, "An Empirical Study About Search-Based Refactoring Using Alternative Multiple and Population-Based Search Techniques," in *Computer and Information Sciences II*, E. Gelenbe, *et al.*, Eds., ed: Springer London, 2012, pp. 59-66.

[163] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution-Research and Practice,* vol. 20, pp. 345-364, Sep-Oct 2008.

[164] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software & Systems Modeling,* vol. 9, pp. 141-160, April 1, 2010 2010.

[165] Y. Khan and M. El-Attar, "Using model transformation to refactor use case models based on antipatterns," *Information Systems Frontiers,* pp. 1-34, 2014/08/13 2014.

[166] R. Seidl, "Modeling Metrics for UML Diagrams," in *Richard Seidl,* ed.

[167] M. Misbhauddin and M. Alshayeb, "Towards a Multi-view Approach to Model-Driven Refactoring," in *2012 African Conference on Sofware Engineering and Applied Computing (ACSEAC)*, 2012, pp. 60-66.

Vitae

Name	:Abdulrahman Ahmed Bobakr Baqais
Nationality	:Yemeni
Date of Birth	:9/29/1980
Email	:baqais@kfupm.edu.sa
Address	:Dhahran- KFUPM Blvd.
Academic Background	:

Mr.Abdulrahman has earned his Bachelor in Information Technology (Hons) Majoring in Software Engineering from Multimedia University, Malaysia in 2007 with a GPA of 3.9 out of 4.0. He earned Master in Science from Staffordshire University –UK in 2010 awarding distinction grade in his Dissertation. He has won a best paper award in 2016 in Italy and a merit award in 2013 in UK. In addition, he won two awards in the third and fifth Saudi scientific conference. He has published more than 10 papers in software engineering field, AI algorithms and computer architecture.

Publications:

1. Abdulrahman Baqais, Mohammad Al-shayeb, Automatic Refactoring of Single and Multi-View UML Models using Artificial Intelligence algorithms, 4th international conference on Model-Driven Engineering and Software Development (MODLELSWARD'2016), Rome, Italy,19-21 Feb,2016.
   (Best Paper Award in Doctorate Consortium Area)
2. M. Niazi, S. Mahmood, M. Alshayeb, A. Baqais, and A. Gill, Motivators and De-Motivators of adopting Social Computing in Global Software Development: Initial Results, The World Congress on Engineering, London, U.K., 3-5 July, 2013
   (Merit-paper Award) (Citation Count: 4)

3. Baqais, A., Assayony, M., Khan, A., & Al-Mouhamed, M., Bank Conflict-Free Access for CUDA-Based Matrix Transpose Algorithm on GPUs, International Conference on Computer Applications Technology,January,2013.
   (Citation Count: 2)

4. Mayez Al-Mouhamed, Allam Fatayer, Anas Almousa, Abdulrahman Baqais, Mohammed Assayony, Padding Free Bank Conflict Resolution for CUDA-Based Matrix Transpose Algorithm, International Journal of Networked and Distributed Computing 2 (3), 124-134,2014
5. Abdulrhman Baqais**,** Genetic Algorithm for function approximation: An Experimental Investigation, International Journal of Artificial Intelligence and Applications, 2016.
6. A. H. Khan, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. Baqais, and M. Assayony, Padding Free Bank Conflict Resolution for CUDA-Based Matrix Transpose Algorithm, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'14), 2014.
7. A. Baqais, M. Alshayeb, and Z. Baig, Hybrid Intelligent Model for Software Maintenance Prediction, The World Congress on Engineering, London, U.K., 3-5 July, 2013 **(Citation Count: 2)**
8. Baqais, A. A. B., Ahmed, M., Sharqawy, M. H., Function Approximation of Seawater Density Using Genetic Algorithm. In Proceedings of the World Congress on Engineering (Vol. 2),2013.
9. A.Baqais, M. Ahmed, Software Cloning Detection Techniques: Comparison Criteria, international conference on software engineering research and practice (SERP'12), USA, July, 2012.
10. Abdulrahman Baqais, Abdullah Owayadh, Using Transformation Language to reconstruct Use Cases using Anti-Pattern, 28th International Conference on Science, Technology and Management (ICSTM), Riyadh, Saudi Arabia, 2015.
11. Abdurlahamn Baqais, Virtualization From Single to Nested Layer, 28th International Conference on Science, Technology and Management (ICSTM), Riyadh, Saudi Arabia, 2015.
12. Abdulrahman Baqais, Mohammad Amro, Mohammad Alshayeb, A closer look on the correlation between stability and maintainability class metrics, The 7th International Conference of Computer Science & Information Technology, 2016

Submitted Papers:

1. M. Niazi, S. Mahmood, M. Alshayeb, A. Baqais, and A. Gill, Motivators of adopting Social Computing in Global Software Development: Initial Results", Journal of Software: Evolution and Process, 2015.
   "Passed Second Review", **(ISI Journal).**

2. Abdulrahman Baqais, A Multi-View Comparison of Various Metaheuristic and Machine Learning Algorithms, International Journal of Soft Computing and Software Engineering, 2016.

3. Abdulrahman Baqais**,** Mohammad Alshayeb, A comparison between three machine learning algorithms in predicting model smells at the class level for object-oriented software, 2016. **(ISI Journal).**
4. Abdulrahman Baqais, Mohammaed Alshayeb, Hybridization of Simulated Annealing and Clustering: A Case Study of Sequence Diagram Refactoring, 2016 **(ISI Journal).**
5. Abdulrahman Ahmed Bobakr Baqais**,** Mohammad Alshayeb, On Single and combined metrics Use Case Refactoring using Search-based Algorithms, 2016 **(ISI Journal).**

Complete Draft:

1. Abdulrahman Ahmed Bobakr Baqais, Mohammad Alshayeb, Systematic literature review on automatic refactoring , 2016
2. Basem Almadani, Abdulrahman Baqais**,** Mohammad Alsaeedi, Framework for Real-Time Distributed Education System, 2016.
3. Abdulrahman Ahmed Bobakr Baqais, Using Social Network to Enhance Collaboration Among Research Students in Higher Education, 2016
4. Abdulrahman Ahmed Bobakr Baqais, Mohammad Alshayeb, Multiple-View Diagram Refactoring using Search-Based Algorithms, 2016
5. Abdulrahman Ahmed Bobakr Baqais, Binary & Real Genetic Algorithms for Seawater Desalination, A Case study of a highly constrained multi-modal strong epistasis engineering problem: Application and Performance Comparison, 2016.

In Progress:

1. Mohammad Yahya, Abdulrahman Ahmed Bobakr Baqais, Mohammad Alshayeb. An Empirical Study of Evaluating the Correlation between Class Stability and Bad Smells, 2016
2. Abdulrahman Ahmed Bobakr Baqais, Moataz Ahmed, Mostafa Sharqawi, A New Correlation for Seawater and Pure Water Density at Different Temperatures, Salinities, and Pressures, 2016
3. Abdulrahman Ahmed Bobakr Baqais, UML Refactoring and Search-Based Algorithms: Mapping Framework, 2016
4. Abdulrahman Ahmed Bobakr Baqais, Using WhatsUp as a collaboration tool in the classroom: A Computer Programming Course Experience, 2016.
5. Abdulrahman Ahmed Bobakr Baqais, Systematic Literature Review in Security Software: A tertiary Study, 2016.

Workshops Attended:

- Experiences in creating excitement in the classroom through active learning techniques", Workshop, 11 Feb,2016
- Enhancing Research Skills , Seminar, 4 March, 2015
- Instructional Objectives Workshop, 12 Nov, 2014
- Active Learning Strategies Workshop, 20 Aug ,2014.
- Integrated Coursed Design Workshop, 16-18 Aug, 2014.
- The Joy and Responsibility of Teaching Well , Workshop, 18 Aug,2014
- A member of Deanship of Academic Development (DAD) projects, KFUPM.
- Institutional Research workshop, June, 2013, KFUPM.
- Student-Centered Active Learning Environments: Design and Implementation workshop ,May, 2012
- Student motivation workshop, 2012

Co-Curriculum and Academic Activities:

- ABET Committee Program 2011-2012.
- Voluntarily participated in KFUPM Major Selection Day , KFUPM, 2012.
- Organizer in Islamic Art and Architecture Event - History Society, Malaysia, 2003 .
- Participating in Entrepreneurship seminars, Malaysia, 2004 – 2005
- Participating in different seminars and workshops regarding writing and publishing research papers, 2008-2012.

Projects:

1- **Optimization of seawater properties for water desalination based on Genetic Algorithm (MIT – KFUPM Collaboration Project) – (2012)**
The aim of this project is to develop a genetic algorithm that draws the best curve for density seawater with fewer numbers of coefficients.

2- **Software Stability Metrics validation (Funded Project) – (2012)**
A new stability metric has been proposed by KFUPM. And the objective of this project is validating the results on various software engineering projects.

3- **Using social network to enhance collaboration in distributed software environment – (2012)**
Conducting a systematic literature review on the motivation factors of adopting social networks in global software environment and an empirical study was performed after the review.

4- **Arabic Handwritten recognition (Funded Project) – Completed (2011).**

Classifying and organizing thousands of images and forms. Validating the results of applying recognition algorithms to the images.

Professional Society Affiliation:

8.1 IEEE, ACM